

# CS213/293 Data Structure and Algorithms 2024

## Lecture 14: Graphs - Depth-first search

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-10-14

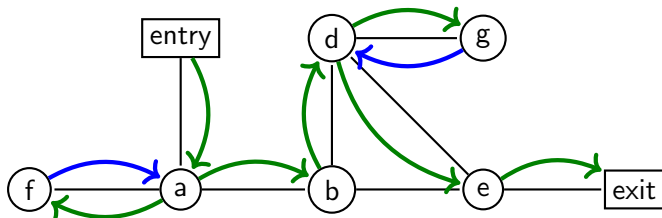
# Topic 14.1

## Depth-first search (DFS)

## Let us solve the maze again

Breadth-first search is about considering all available options before exploring further.

We can have another strategy of search: explore a **choice exhaustively** before considering another choice.



## Algorithm: DFS for search

---

**Algorithm 14.1:** DFS( Graph  $G = (V, E)$ , vertex  $r$ , Value  $x$  )

---

```
1 Stack S;  
2 set visited;  
3 S.push(r);  
4 while not S.empty() do  
5      $v := S.pop()$ ;  
6     if  $v.label == x$  then  
7         return  $v$   
8     if  $v \notin visited$  then  
9          $visited := visited \cup \{v\}$ ;  
10        for  $w \in G.adjacent(v)$  do  
11            S.push(w)
```

---

## Example: DFS

Green vertices in the  $S$  are already visited vertices and the first unvisited vertex is processed next.

Initially:  $S = [\text{entry}]$

After visiting entry:  $S = [a]$

After visiting a:  $S = [f, b, \text{entry}]$

After visiting f:  $S = [a, b, \text{entry}]$

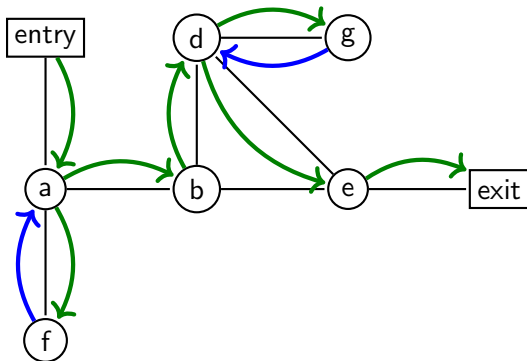
After visiting b:  $S = [a, d, e, \text{entry}]$

After visiting d:  $S = [b, g, e, e, \text{entry}]$

After visiting g:  $S = [d, e, e, \text{entry}]$

After visiting e:  $S = [\text{exit}, b, d, e, \text{entry}]$

After visiting exit: Node is found.



### Exercise 14.1

Is there a bound that limits the size of  $S$ ?

If any node is popped out, then only we'll visit add its neighbor so max  $n-1$  can be inserted.

# Algorithm: Recursive DFS

The recursive description of DFS is easier to follow.

---

**Algorithm 14.2:** DFS( graph  $G = (V, E)$ , vertex  $v$  )

---

```
1 for  $v \in V$  do
2    $v.visited := False$ 
3 DFSREC( $G, v$ )
```

---

---

**Algorithm 14.3:** DFSREC( Graph  $G$ , vertex  $v$  )

---

```
1  $v.visited := True$ ;
2 for  $w \in G.adjacent(v)$  do
3   if  $w.visited == False$  then
4     DFSREC( $G, w$ )
```

---

$v$  can be in three possible states

- ▶  $v$  is not visited
- ▶  $v$  is on the call stack
- ▶  $v$  is visited and not on the call stack

## Exercise 14.2

Why is there no stack in the recursive DFS? Recursion goes into the call stack.

# DFS Tree

---

**Algorithm 14.4:** DFS( graph  $G = (V, E)$ , vertex  $v$  )

---

```
1 global time := 0;  
2 for  $v \in V$  do  
3    $v.visited := False$   
4 DFSREC( $G, v$ )
```

---

---

**Algorithm 14.5:** DFSREC( Graph  $G$ , vertex  $v$  )

---

```
1  $v.visited := True$ ;  
2  $v.arrival := time++$ ;  
3 for  $w \in G.adjacent(v)$  do  
4   if  $w.visited == False$  then  
5      $w.parent := v$ ;  
6     DFSREC( $G, w$ )  
7  $v.departure := time++$ ;
```

---

## Example: recursive execution

Green numbers are arrival times and blue numbers are the departure times.

DFSREC( $G, entry$ )

DFSREC( $G, a$ )

DFSREC( $G, f$ )

Exit DFSREC( $G, f$ )

DFSREC( $G, b$ )

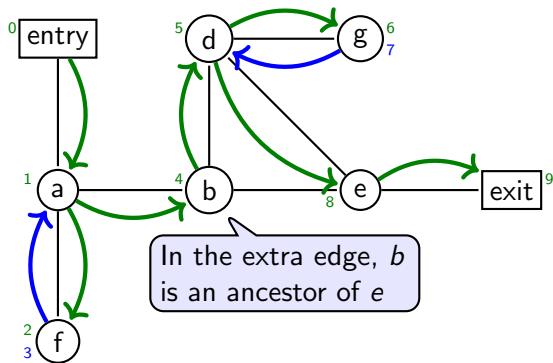
DFSREC( $G, d$ )

DFSREC( $G, g$ )

Exit DFSREC( $G, g$ )

DFSREC( $G, e$ )

DFSREC( $G, exit$ )



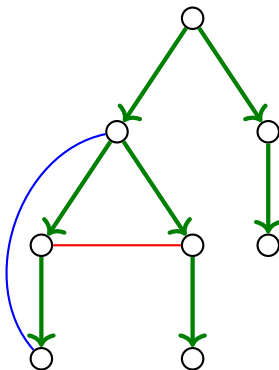
### Exercise 14.3

What are the exit timings of the remaining nodes?



## Non-tree edges

A run of DFS induces a tree. There are two kinds of possible extra edges.



### Exercise 14.4

- Is blue edge possible?*
- Is red edge possible?*

Yes.  
No.

# Reachable coverage

## Theorem 14.1

Let  $G = (V, E)$  be a connected graph. For each  $\{v_1, v_2\} \in E$ ,  $v_1$  is an ancestor of  $v_2$  in DFS tree or vice versa.

### Proof.

Without loss of generality, we assume  $v_1.\text{arrival} < v_2.\text{arrival}$  at the end of DFS.

During the run of  $\text{DFSREC}(G, v_1)$ ,  $v_2$  will be visited in one of the following two ways.

1.  $\text{DFSREC}(G, v_2)$  is called by  $\text{DFSREC}(G, v_1)$ .  $v_1$  is the parent of  $v_2$ .
2.  $\text{DFSREC}(G, v_2)$  has been called already, when the loop in  $\text{DFSREC}(G, v_1)$  reaches to  $v_2$ .

In either case,  $v_2.\text{arrival} < v_1.\text{departure}$ . Therefore,  $v_1$  is an ancestor of  $v_2$  in the DFS tree.

(Why?)

Upon reaching to a node, when we explore its adjacency list and we find a node already visited then during dfs call

In case 1, we call  $\{v_1, v_2\}$  a tree edge. In case 2, we call  $\{v_1, v_2\}$  a back edge.

# Running time of DFS

## Theorem 14.2

*The running time of DFS is  $O(|E| + |V|)$ .*

### Proof.

The total number of recursive calls and iterations of initializations is  $O(|V|)$ .

In call  $DFSRec(G, v)$ , the loop iteration is bounded by  $degree(v)$ .

Therefore, the total number of iterations is  $O(|E|)$ .

Iteration over nodes in adjacency list(for loop), and for each node a dfs call.

Therefore, the running time is  $O(|E| + |V|)$ .

in directed graph all edges will be explored once and twice in case of undirected graph.



## Exercise 14.5

*Prove that the running time besides initialization is  $O(|E|)$ .*

Whether a node is visited already or not you'll visit each node once if visited then continue and if not then initialize.

## Algorithm: DFS for not connected graph

---

**Algorithm 14.6:** DFS<sub>FULL</sub>( graph  $G = (V, E)$  )

---

```
1 global time := 0;  
2 for  $v \in V$  do  
3    $v.visited := False$   
4 while  $\exists v$  such that  $v.visited == False$  do  
5   DFSREC( $G, v$ )
```

---

## Parent relation is a forest

A forest is a collection of disjoint trees, which occurs when DFS is run on a graph which has more than one connected component.

A spanning tree of a graph is a tree that includes all the vertices of the graph and is a subgraph that connects all the vertices with the minimum number of edges.(i.e.  $|V|-1$  for  $V$  vertex)

### Theorem 14.3

*The parent relation after the run of  $\text{DFS}_{\text{FULL}}(\text{graph } G = (V, E))$  induces spanning trees over a connected components of  $G$ .*

### Proof.

Each call to  $\text{DFS}_{\text{REC}}(G, v)$  will traverse a connected component of  $G$  that contains unvisited node  $v$ . (Why?)

If the component has  $k$  nodes, then the tree has  $k - 1$  edges, because the parent of  $v$  will be Null.

Therefore, the parent relation is a tree over the component.



## Topic 14.2

Does the graph have a cycle?

## Detecting cycle

If there is a back edge, there is a cycle. We modify our DFS<sub>REC</sub> as follows.

---

**Algorithm 14.7:** DFS<sub>REC</sub>( Graph  $G$ , vertex  $v$  )

---

```
1  $v.arrival := time++$ ;  
2 for  $w \in G.adjacent(v) - \{v.parent\}$  do  
3   if  $w.visited == False$  then  
4      $w.parent := v$ ;      If it is already visited earlier then that means it has a back-edge in dfs call  
5     DFSREC( $G, w$ )  
6   else  
7     raise "Found Cycle";  
8  $v.departure := time++$ ;
```

---

## Back edge == cycle

### Theorem 14.4

*A graph  $G$  has a cycle iff  $\text{DFSFull}(G, v)$  has a back edge for some  $v \in G$ .*

### Proof.

#### **forward direction:**

Due to theorem 14.3, each call to  $\text{DFSREC}$  without exception will produce a spanning tree over a connected component of  $G$ .

Since there are no extra edges besides the tree, the cycle cannot be formed within the component.

#### **reverse direction:**

If the exception "Found cycle" is raised, there are two paths between  $u$  and  $w$ .

- ▶ Edge  $\{u, w\}$ .
- ▶ path via the parent relation that does not contain  $\{u, w\}$ .

Therefore, we have a cycle. □



## Topic 14.3

### Checking 2-edge connected graphs

Any edge removed from a graph; we still have scope to reach all the nodes in the graph.

## 2-edge connected graph

### Definition 14.1

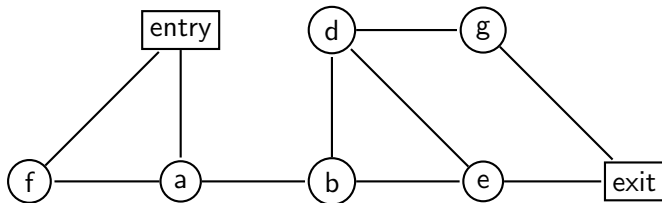
A graph  $G = (V, E)$  is *2-edge connected* if for each  $e \in E$ ,  $G - \{e\}$  is a connected graph.

2-edge connected graphs are useful for designing resilient networks that are tolerant of link failures.

## Example: 2-edge connected graphs

### Example 14.1

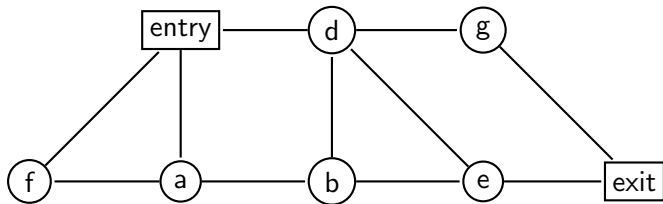
The following graph is not 2-edge connected.  $\{a, b\}$  is called *bridge*.



## Example: 2-edge connected graphs (2)

### Example 14.2

*The following graph is 2-edge connected.*



## Näive algorithm for checking 2-edge connectivity

For each edge, delete the edge and check connectedness.

The algorithm will run in  $O(|E|^2)$ .

We are looking for something more efficient.

## Idea: 2-edge connectivity via DFS

Observation 1: If we delete any number of back edges, the graph remains connected.

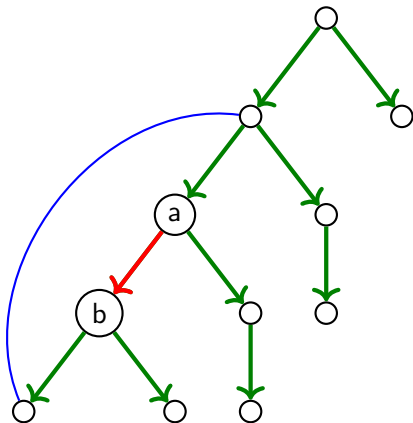
Observation 2: If a tree edge is part of some cycle, the graph remains connected after its deletion.

How can we check this?

# Checking participation in a cycle

## Example 14.3

*The red edge  $(a, b)$  in the following DFS tree is part of a cycle if there is a back edge that starts at one of the decedents of  $b$  and ends at an ancestor of  $a$ .*



## Deepest back edge

We need to track the back edges that cover most edges.

### Definition 14.2

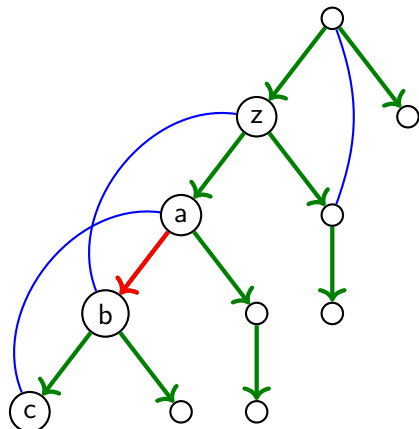
*The deepest back edge for a vertex is the back edge that goes from the descendent of the vertex to an ancestor of the lowest level.*

### Example 14.4

*In the following DFS tree, there are two back edges from the decedents of  $b$ .*

*$\{b, z\}$  is deeper back edge than  $\{c, a\}$ .*

*Since there is no other back edge from decedents of  $b$ ,  $\{b, z\}$  is the deepest back edge for  $b$ .*



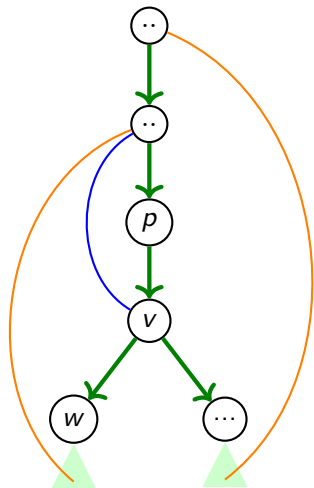


## How do we identify the deepest back edge?

By comparing the arrival times of the destinations, we identify the deepest back edge.

We consider all neighbors of vertex  $v$  to find the deepest back edge. There are three possible cases.

1. child on DFS tree: recursively find the deepest edge
2. parent on the DFS tree: to be ignored
3. back edge: candidate for the deepest edge



---

**Algorithm 14.8:** `int 2EC( Graph G, vertex v )`

---

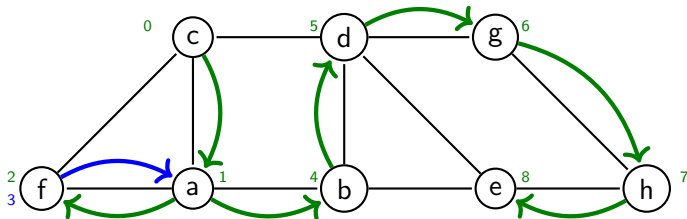
```
1 v.visited := True;
2 v.arrival := time ++;
3 deepest := v.arrival;
4 for  $w \in G.\text{adjacent}(v) - \{v.\text{parent}\}$  do
5     if w.visited == False then
6         w.parent := v;
7         deepest' := 2EC(G, w);
8     else
9         deepest' := w.arrival;
10    deepest := min(deepest, deepest'); This ensures that lowest leveled node gives us the back-edge.
11 v.departure := time ++;
12 if v.parent  $\neq$  Null and v.arrival == deepest then raise "Bridge found!";
13 return deepest;
```

---

## Example: 2-edge connectedness

### Exercise 14.6

Consider the following DFS run of the following graph.



What are the deepest back edges for the following nodes?

► e d

► h g

► g d

► d c

► b a

► f c

► a c

► c c

## Topic 14.4

### Depth-first search for directed graph

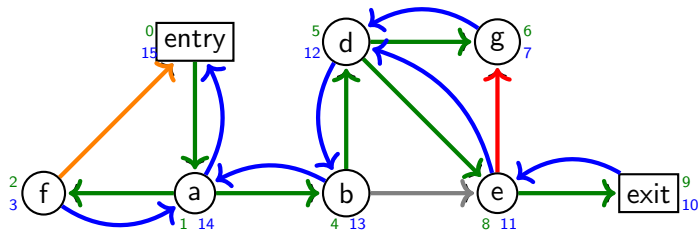
## DFS for directed graph

There is no change in the code of `DFSFULL` for the directed graph, the code will work as it is.

However, some of the behavior concerning extra edges will change.

## Example : DFS on the directed graph

Consider the following directed graph.



Now we have three kinds of extra edges.

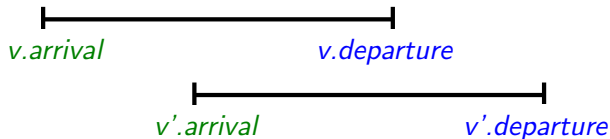
- ▶ Forward edge:  $(b, e)$ , where  $b.arrival < e.arrival < e.departure < b.departure$
- ▶ Back edge:  $(f, entry)$ , where  $entry.arrival < f.arrival < f.departure < entry.departure$
- ▶ Cross edge:  $(e, g)$ , where  $g.arrival < g.departure < e.arrival < e.departure$

Are there other kind of edges?

## Interleaved intervals are not possible

Theorem 14.5 As the LIFO policy it won't be possible.

For each  $v, v' \in V$ ,  $v.\text{arrival} < v'.\text{arrival} < v.\text{departure} < v'.\text{departure}$  is not possible.



Proof.

Let us assume  $v'.\text{arrival}$  is between  $v.\text{departure}$  and  $v.\text{arrival}$ .

Therefore,  $v$  is in the call stack when  $v'$  is put on the call stack during a run of DFS<sub>REC</sub>.

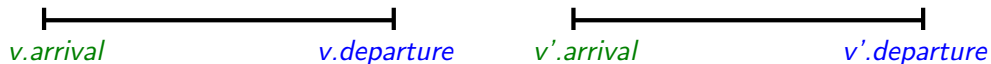
$v'$  will leave the call stack before  $v$ .

Therefore,  $v'.\text{departure} < v.\text{departure}$ . The ordering of events is not possible. □

DFS always follows the available edges.

### Theorem 14.6

For each  $(v, v') \in E$ ,  $v.\text{arrival} < v.\text{departure} < v'.\text{arrival} < v'.\text{departure}$  is not possible.



Even in the case of directed graphs,  $v.\text{arrival}$  and  $v.\text{departure}$  will be more than  $v'(\text{arr}+\text{dep})$  like the curr ordering will be reversed.

### Proof.

Apply theorem 14.1 after replacing the undirected edges with the directed edges in the theorem. □

### Exercise 14.7

- Prove theorem 14.1 for directed graph.
- The theorem was proven for DFS<sub>REC</sub>. Extend it for DFS<sub>FULL</sub>.

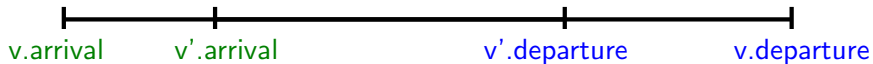
**Commentary:** We need to reword the theorem to show that  $v'$  will be on the call stack before  $v$  departs the call stack.



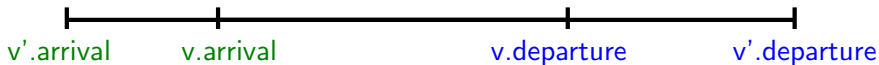
## Extra edges

We are left with only the following possibilities for the extra and tree edges. Let  $(v, v') \in E$ .

- ▶ Forward edge/Tree edge



- ▶ Back edge:



- ▶ Cross edge:



### Exercise 14.8 (Final 2023)

- Show: If  $v$ .departure  $\leq v'$ .departure,  $(v, v')$  is a back edge. When  $v$  departs  $v$  is still in call stack.
- Give the condition that identifies the back or cross edge.

## Topic 14.5

Does the directed graph have a cycle?

Idea: Back edge == cyclic

If DFS finds a back edge there is a cycle in a (directed) graph.

### Exercise 14.9

*How can we use BFS to find cycles?*

# Algorithm: Has Cycle?

---

**Algorithm 14.9:** HASCYCLE( directed graph  $G = (V, E)$  )

---

```
1 Let  $v \in V$ ;  
2 DFSFULL( $G, v$ );  
3 if  $\exists (v, v') \in E$  such that  $v.\text{departure} \leq v'.\text{departure}$  then  
4   return True;  
5 return False;
```

Condition given is enough to say that it has a back-edge.  
or make to maps visited and dfs visited which will be true when visiting and false after we are done business with graph.

---

## Back edge == Cycle

### Theorem 14.7

*A directed graph  $G = (V, E)$  has a cycle iff  $DFSFull(G)$  has a back edge.*

**Proof.**

**forward direction:**

Suppose there is no back edge. Therefore,  $\forall (v, v') \in E, v.departure > v'.departure$ .

Sort all the nodes by their departure times.

All edges will be going in one direction of the sorted sequence. Therefore, there is no cycle.

**reverse direction:**

Let us suppose there is a back edge  $(v, v') \in E$ . Therefore,  $v.departure \leq v'.departure$ .

Due to properties of the back edge,  $v'$  must be on the call stack when  $v$  departs.

Therefore, there is a path from  $v'$  to  $v$ . Therefore, there is a cycle. □

**Commentary:** Does the above argument work when  $v$  and  $v'$  are equal?

## Topic 14.6

### Topological sort

# Topological order

## Definition 14.3

*For a DAG  $G = (V, E)$ , the topological order  $<$  is an order of vertices of  $V$  such that if  $(v, v') \in E$  then  $v < v'$ .*

## Algorithm: topological sort

---

**Algorithm 14.10:** TOPOLOGICALSORT( directed graph  $G = (V, E)$  )

---

- 1 DFSFULL( $G, v$ );
  - 2 **if**  $\exists (v, v') \in E$  such that  $v.\text{departure} \leq v'.\text{departure}$  **then**
  - 3     **return** "Cycle found: Sorting not possible";
  - 4 **return** sorted vertices of  $V$  in the decreasing order of  $\text{departure}$ .
- 

### Exercise 14.10

*Can we avoid sorting after the DFS run?*

There is an incorrect order of elements in the sorted array.



## Topic 14.7

Is strongly connected?

## Strongly connected (Recall)

If the graph has no two or more component, then it is strongly connected in indirect graph.

Consider a directed graph  $G = (V, E)$ .

### Definition 14.4

$G$  is *strongly connected* if for each  $v, v' \in V$  there is a path  $v, \dots, v'$  in  $E$ .

# Näive algorithm

Run DFS from each vertex, and check if all vertices are reached.

The running time complexity is  $O(|V||E|)$ .

## Strongly connected via DFS

Condition 1: If  $DFS(v)$  visits all vertices in  $G$  then there is a path from  $v$  to each vertex in  $G$ .

Condition 2: There is a path from every node in  $G$  to  $v$ .

We can check condition 1 using DFS 1. How can we check condition 2?

# Kosaraju's algorithm

Run DFS on  $G$  and  $G^R$  (All edges of  $G$  are reversed) from some vertex  $v$ .

If BOTH DFSs cover all nodes,  $G$  is strongly connected.

The running time complexity is  $O(|V| + |E|)$ .

## Exercise 14.11

*Can we use BFS here?* Kosaraju's algorithm heavily depends on DFS hence it can't be implemented using bfs.

Let us see if we can avoid two passes of the graph.

# How can we check if we can reach the root of DFS?

## Example 14.5

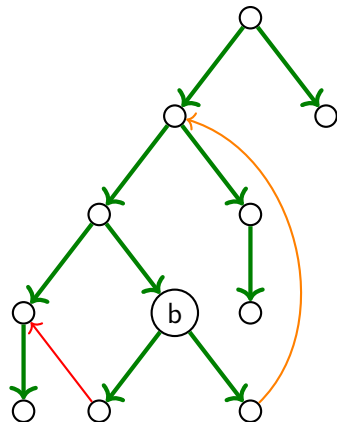
Consider vertex  $b$ . We must be able *to escape the subtree of  $b$*  to reach the root.

There are only two ways to escape a subtree.

► Back edge

► Cross edge

Can both routes to escape be useful?



## Lower arrival times

Observation: both back and cross edges take us to lower arrival times.

Induction: If we can escape from the subtree of each vertex, we are guaranteed to reach the root.

All we have to do is to show that we can escape from each vertex.

Use earliest escape edge? (Same idea as 2EC?)

## Algorithm: SC

---

**Algorithm 14.11:** `int SC( Graph G, vertex v )`

---

```
1 v.visited := True;
2 v.arrival := time ++;
3 earliest := v.arrival;
4 for w ∈ G.adjacent(v) do
5     if w.visited == False then
6         w.parent := v;
7         earliest' := SC(G, w);
8     else
9         earliest' := w.arrival;
10    earliest := min(earliest, earliest');
11 v.departure := time ++;
12 if v.parent ≠ Null and v.arrival == earliest then raise "Not strongly connected!";
13 return earliest;
```

---



## Topic 14.8

### Tutorial problems

## Exercise: 2-vertex connected graph

### Definition 14.5

A graph  $G = (V, E)$  is *2-vertex connected* if for each  $v \in V$ ,  $G - \{v\}$  is a connected graph.

### Exercise 14.12

*Give an algorithm that checks if a graph is 2-vertex connected.*

## Exercise: Change in DFS

Suppose that we have a graph and we have run DFS starting at a vertex  $s$  and obtained a DFS tree. Next, suppose that we add an edge  $(u,v)$ . Under what conditions will the DFS tree change? Give examples.

## 2-edge-connectedness in theory

### Exercise 14.13

Let  $G(V, E)$  be a graph. We define a relation on edges as follows: two edges  $e$  and  $f$  are related if there is a cycle containing both (denoted by  $e \equiv f$ ).

1. Show that this is an equivalence relation. The equivalence class  $[e]$  of an edge  $e$  is called its connected component.
2. What is the property of the equivalence relation when we say the graph is 2-edge connected?

## Exercise: SCCs via Kosaraju's algorithm

### Exercise 14.14

*Modify Kosaraju's algorithm to identify all SCCs of a graph.*

### Exercise 14.15

*Give similar modifications for the algorithm SC.*

## Exercise: Classification of edges via BFS

### Exercise 14.16

*If we run BFS on a directed graph, can we define the same classes of edges, i.e., cross, tree, back, and forward edges? Give conditions for each class.*

# Detecting cycle during the run for a directed graph!

## Exercise 14.17

Let us modify DFS<sub>REC</sub> to detect cycles during the run. Give the expression for the condition to detect the cycles.

---

**Algorithm 14.12:** DFS<sub>REC</sub>( Graph  $G$ , vertex  $v$  )

---

```
1  $v.visited := True$ ;  
2  $v.arrival := time++$ ;  
3 for  $w \in G.adjacent(v)$  do  
4   if  $w.visited == False$  then  
5     DFSREC( $G, w$ )  
6   else  
7     if condition then  
8       throw "Found Cycle"  
9  $v.departure := time++$ ;
```

---

# Topic 14.9

## Problems



## Exercise: another search

### Exercise 14.18

We have a new search algorithm that uses a set  $S$  for which we have two functions (i)  $\text{ADD}(x, S)$  which adds  $x$  to  $S$ , and (ii)  $y = \text{SELECT}(S)$  which returns an element of  $S$  following a certain rule and removes  $y$  from  $S$ .

---

**Algorithm 14.13:** `int MYSEARCH( Graph  $G = (V, E)$ , vertex  $s$  )`

---

```
1 for  $v \in V$  do  $v.visited = \text{False}$ ; bfs,  $S$  is queue and dfs  $S$  is stack. In bfs most likely be a direct edge since it  
2 for  $e \in E$  do  $e.found = \text{False}$ ; prioritize the nearest edge to source. dfs, it can be any edge since dfs  
3  $S = \text{empty}$ ;  $\text{ADD}(s, S)$ ;  $nos := 1$ ;  $\text{record}[nos] := s$ ; explores one way till end before exploring the other.  
4 while not  $S.\text{empty}()$  do  
5      $s := \text{select}(S)$ ;  $nos := nos + 1$ ;  $\text{record}[nos] = y$ ;  
6     for  $w \in G.\text{adjacent}(v)$  do  
7         if  $w.visited == \text{False}$  then  $w.visited := \text{True}$ ;  $\text{found}[\{u, v\}] = \text{true}$ ;  $\text{ADD}(v, S)$ ;
```

---

1. Compare the bfs and dfs algorithms with the above code. Take special care in understanding visited.
2. Let us look at the sequence  $\text{record}[1], \text{record}[2], \dots, \text{record}[n]$ . Show that there is a path from  $\text{record}[i]$  to  $\text{record}[i+1]$  using only edges that have been found at that point.
3. Compare BFS and DFS in terms of the above path lengths.

## Topic 14.10

Extra slides: Topological sort

## Algorithm: Topological sort using DFS

An implementation with cycle detection and in-place sorting.

---

**Algorithm 14.14:** TOPOLOGICALSORT( Directed graph  $G = (V, E)$  )

---

```
1 Stack Sorted;  
2 while  $\exists v$  such that  $v.visited == False$  do visit( $v$ ) ;
```

---

---

**Algorithm 14.15:** VISIT( Graph  $G = (V, E)$ , vertex  $v$  )

---

```
if  $v.visited$  then return;  
if  $v.onPath$  then throw Cycle found ;  
 $v.onPath = True$ ;  
for  $w \in G.adjacent(v)$  do  
    | VISIT( $w$ )  
 $v.onPath = False$ ;  
 $v.visited = True$ ;  
Sorted.push( $v$ );
```

---

End of Lecture 14