CS213/293 Data Structure and Algorithms 2025

Lecture 15: Graphs - Depth-first search

Instructor: Ashutosh Gupta

IITB India

Compile date: October 30, 2025

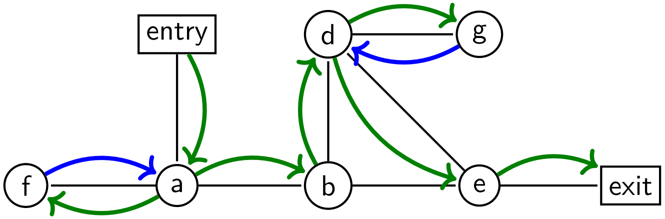
Topic 15.1

Depth-first search (DFS)

Let us solve the maze again

Breadth-first search is about exploring all available options before exploiting an option further.

Another strategy of search: exploit a choice exhaustively before exploring another choice.



Algorithm: DFS for search

Algorithm 15.1: DFS(Graph $G = (V, E)$, vertex r, Value x)

```
Stack S;
set $visited$;
$S.push(r)$;
while not $S.empty()$ do
    $v := S.pop()$;
    if $v.label == x$ then
        return $v$
    if $v \notin visited$ then
        $visited := visited \cup \{v\}$;
        for $w \in G.adjacent(v)$ do
            $S.push(w)$
```

Example: DFS

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

Initially: $S = [entry]$

After visiting entry: $S = [a]$

After visiting a: $S = [f, b, entry]$

After visiting f: $S = [a, b, entry]$

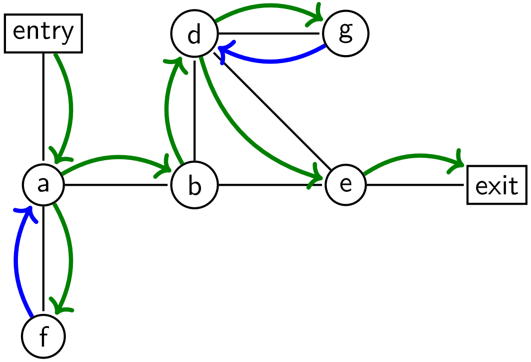
After visiting b: $S = [a, d, e, entry]$

After visiting d: $S = [b, g, e, e, entry]$

After visiting g: $S = [d, e, e, entry]$

After visiting e: $S = [exit, b, d, e, entry]$

After visiting exit: Node is found.



Green nodes in S are visited.

Exercise 15.1

Is there a bound that limits the size of S?

Algorithm: Recursive DFS

The recursive description of DFS is easier to follow.

Algorithm 15.2: DFS(graph $G = (V, E)$, vertex v)

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

Algorithm 15.3: 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 15.2

Why is there no stack in the recursive DFS?

Means of analysis of DFS

Recall: we used the concept of “BFS tree” and “level” to analyze the behavior of BFS algorithm.

For DFS, we also collect three pieces of auxiliary information.

- DFS Tree,
- Arrival times, and
- departure times in the stack.

DFS Tree and auxiliary information

Algorithm 15.4: DFS(graph $G = (V, E)$, vertex v)

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

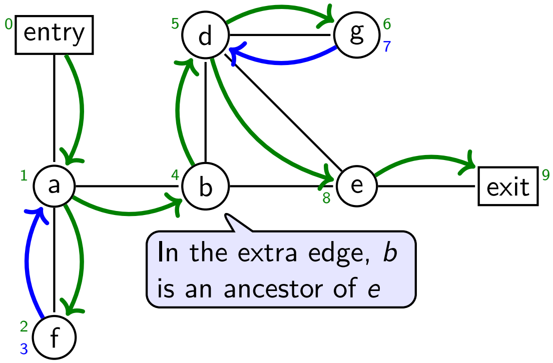
Algorithm 15.5: DFSRec(Graph G, vertex v)

```
1 $v.visited := True$;
2 $v.arrival := time++$;                    //Collecting auxiliary information!
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($G$,$exit$)
```

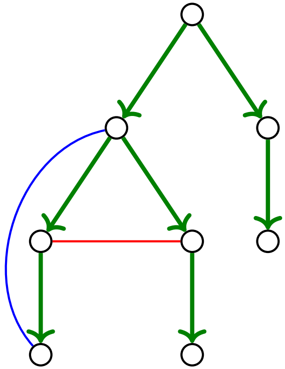


Exercise 15.3

What are the exit timings of the remaining nodes?

Non-tree edges

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



Commentary: We are drawing DFS tree differently from the BFS tree. In BFS lecture, the arrows were pointing in the reverse direction, which indicate the parent pointers in the nodes. In DFS, the forward direction the arrows naturally describe the order of recursive calls. However, both the drawings mean a spanning tree.

Exercise 15.4

a. Is blue edge possible? Yes.

b. Is red edge possible? No.

Reachable coverage

Theorem 15.1

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 $v_1.\mathit{arrival} < v_2.\mathit{arrival}$ at the end of DFS.

During the run of $\mathrm{DFSR}\textsc{ec}(G, v_1)$, v_2 will be visited in one of the following two ways.

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

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

(Why?) $\square$

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

Commentary: Answer to the above why: v_1 was in the call stack when v_2 arrived. The call stack is the ancestor relation.

Running time of DFS

Theorem 15.2

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 $O(|E|)$.

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

Exercise 15.5

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

Algorithm: DFS for not connected graph

Algorithm 15.6: DFSFull(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

Theorem 15.3

The parent relation after the run of $\text{DFSF}\textsc{ull}(\text{ graph } G = (V, E)\ )$ induces spanning trees over the connected components of G.

Proof.

Each call to $\text{DFSR}\textsc{ec}(G, v)$ 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. $\square$

Topic 15.2

Does the graph have a cycle?

Detecting cycle

If there is a back edge, there is a cycle. We modify our DFSREC as follows.

Algorithm 15.7: DFSREC(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         $w.parent := v$;
5         DFSREC($G, w$)
6     else
7         raise "Found Cycle";
8 $v.departure := time++$;
```

We assume the above function will be called by DFSFULL that resets visited bits.

Back edge == cycle

Theorem 15.4

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

Proof.

forward direction:

Due to theorem 15.3, each call to $\mathrm{DFS}\textsc{Rec}$ 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. $\square$

Topic 15.3

Checking 2-edge connected graphs

2-edge connected graph

Definition 15.1

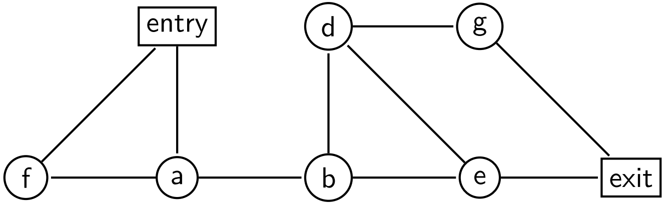
A graph $G = (V, E)$ is 2-edge connected if for each $e \in E$, $G - \{e\}$ is a connected graph.

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

Example: 2-edge connected graphs

Example 15.1

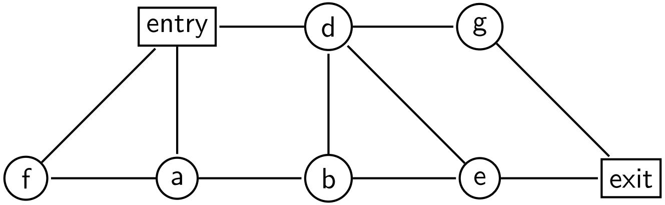
The following graph is not 2-edge connected. $\{a, b\}$ is called bridge.



Example: 2-edge connected graphs (2)

Example 15.2

The following graph is 2-edge connected.



Naive 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.

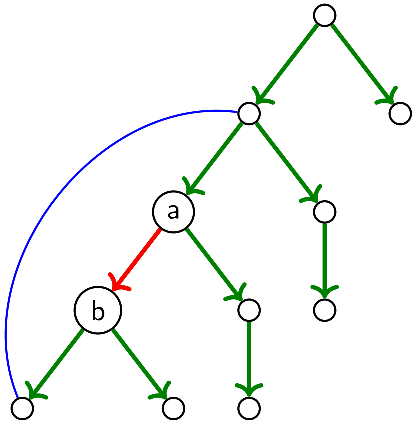
Observation 2: If a tree edge is part of some cycle, the graph remains connected after its deletion.

How can we check this?

Checking participation in a cycle

Example 15.3

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 descendants of b and ends at an ancestor of a.



Deepest back edge

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

Definition 15.2

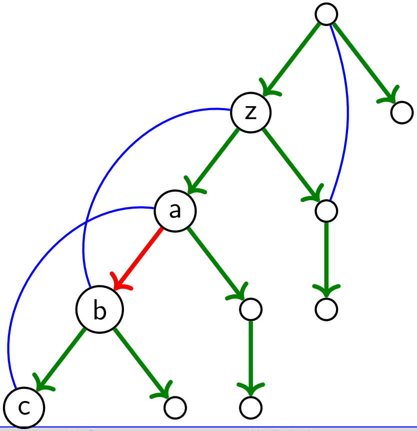
The deepest back edge for a vertex is the back edge that goes from the descendant of the vertex to an ancestor of the lowest level.

Example 15.4

In the following DFS tree, there are two back edges from the descendant of b.

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

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

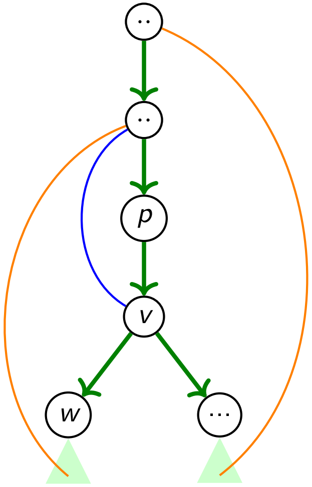


How do we identify the deepest back edge?

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: 2-edge connectedness

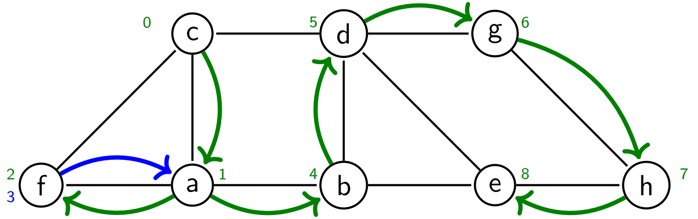
Algorithm 15.8: int $2\mathrm{EC}($ Graph G, vertex v)

```
v.visited := True;
v.arrival := time++;
deepest := v.arrival;
for w ∈ G.adjacent(v) − {v.parent} do
    if w.visited == False then
        w.parent := v;
        deepest' := 2EC(G, w);
    else
        deepest' := w.arrival;
    deepest := min(deepest, deepest');
v.departure := time++;
if v.parent ≠ Null and v.arrival == deepest then raise "Bridge found!";
return deepest;
```

Example: 2-edge connectedness

Exercise 15.6

Consider the following DFS run of the following graph.



What are the deepest back edges for the following nodes?

- *e*
- *h*
- *g*
- *d*
- *b*
- *f*
- *a*
- *c*

Topic 15.4

Depth-first search for directed graph

DFS for directed graph

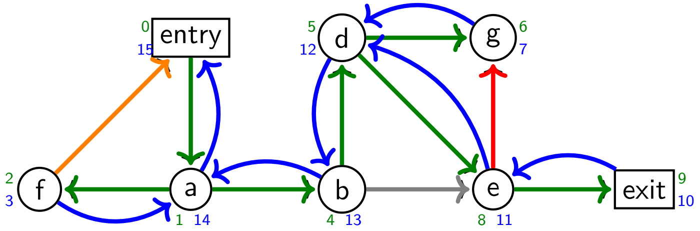
There is no change in the code of $\textsc{DFSFull}$ for the directed graph, the code will work as it is.

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

Example : DFS on the directed graph

Commentary: Green Edges are recursive calls and the blue edges are the return actions. Green superscripts are the arrival times on the call stack. Blue subscripts are the departure times from the call stack.

Consider the following DFS run on a directed graph.



Now we have three kinds of extra edges.

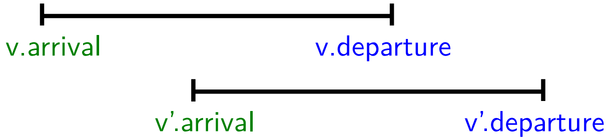
- Forward edge:(b, e), where $b.arrival < e.arrival < e.departure < b.departure$
- Back edge:$(f, entry)$, where $entry.arrival < f.arrival < f.departure < entry.departure$
- Cross edge:(e, g), where $g.arrival < g.departure < e.arrival < e.departure$

Are there other kind of edges?

Interleaved intervals are not possible

Theorem 15.5

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



Proof.

Let us assume $v'.arrival$ is between $v.departure$ and $v.arrival$.

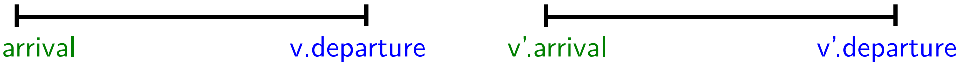
Therefore, v is in the call stack when v' is put on the call stack during a run of DFSRec.

Due to the nature of the recursive calls, v' will leave the call stack before v.

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

DFS always follows the available edges.

Theorem 15.6
For each $(v, v') \in E$, $v.arrival < v.departure < v'.arrival < v'.departure$ is not possible.



Proof.
Apply theorem 15.1 after replacing the undirected edges with the directed edges in the theorem. □

Exercise 15.7
a. Prove theorem 15.1 for directed graph.

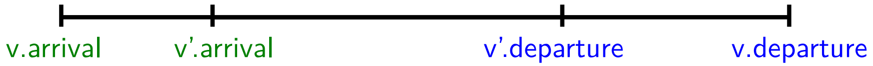
b. The theorem was proven for DFSREC. Extend it for DFSFULL.

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

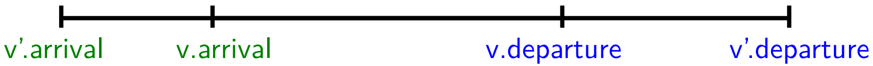
Extra edges

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

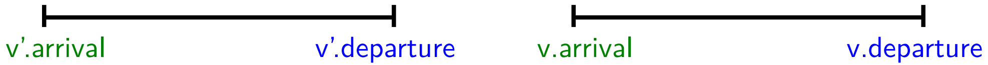
- Forward edge/Tree edge



- Back edge:



- Cross edge:



Exercise 15.8 (Final 2023)

a. Show: If $v.departure \leq v'.departure$, (v, v') is a back edge.

b. Give the condition that identifies the back or cross edge.

Topic 15.5

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 15.9
How can we use BFS to find cycles?

Algorithm: Has Cycle?

Algorithm 15.9: HasCycle(directed graph $G = (V, E)$)

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

Back edge == Cycle

Theorem 15.7

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 15.6

Topological sort

Topological order

Definition 15.3

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 15.10: TOPOLOGICALSORT(directed graph $G = (V, E)$)

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

Exercise 15.10

Can we avoid sorting after the DFS run?

Topic 15.7

Is strongly connected?

Strongly connected (Recall)

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

Definition 15.4

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

Naive 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 15.11
Can we use BFS here?

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

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

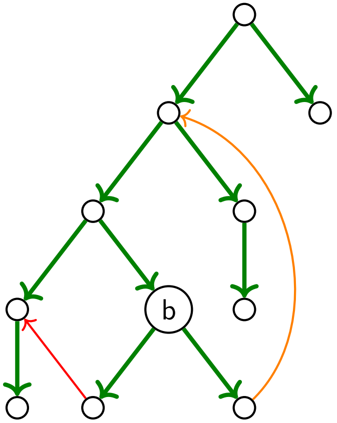
Example 15.5

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 15.11: int SC(Directed Graph G, vertex v)

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

Topic 15.8

Tutorial problems

Exercise: checking visited set

Algorithm 15.12: DFS2(Graph $G = (V, E)$, vertex r, Value x)

```
Stack S;
set $visited$;
$S.push(r)$;
while not $S.empty()$ do
    $v := S.pop()$;
    if $v.label == x$ then
        return $v$
    $visited := visited \cup \{v\}$;
    for $w \in G.adjacent(v)$ do
        if $w \notin visited$ then
            $S.push(w)$
```

Exercise 15.12

Consider another non-recursive version of the DFS in the left. Here, we are checking visited set at the time of pop instead of push in the stack.

1. Is this algorithm correct?
2. In Algorithm 15.12, is it possible a vertex can enter in stack multiple times?
3. Compare the stack behaviors in Algorithms 15.1 and 15.12
4. In algorithm 15.2, is it possible DFSRec is called multiple times for the same vertex?

Exercise: 2-vertex connected graph

Definition 15.5
A graph $G = (V, E)$ is 2-vertex connected if for each $v \in V$, $G - \{v\}$ is a connected graph.

Exercise 15.13
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 15.14

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 15.15

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

Exercise 15.16

Give similar modifications for the algorithm SC.

Exercise: SCCs of the reversed graphs (Final 2024)

Exercise 15.17

Given a directed graph G. Call G^R the graph with all edges reversed (direction). Let $SCC(G)$ be a function that returns the directed graph over the strongly connected components of G, where each strongly connected component is considered a single node. Prove that $SCC(G)^R = SCC(G^R)$.

Exercise: Classification of edges via BFS

Exercise 15.18

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 15.19

Let us modify DFSRec to detect cycles during the run. Give the expression for the condition to detect the cycles.

Algorithm 15.13: 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         DFSRec$(G, w)$
6     else
7         if $condition$ then
8             throw "Found Cycle"
9 $v.departure := time++$;
```

Topic 15.9

Problems

Exercise: another search

Exercise 15.20

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

Algorithm 15.14: int $\textsc{mysearch}$(Graph $G = (V, E)$, vertex s)

```
for $v \in V$ do  $v.visited = False$;
for $e \in E$ do  $e.found = False$;
S=empty;$\textsc{Add}$(s,S);$nos := 1$;$record[nos] := s$;
while $not\ S.empty()$ do
    $v := select(S)$; $nos := nos + 1$; record[nos] = v;
    for  $w \in G.adjacent(v)$ do
        if $w.visited == False$ then  $w.visited := True$; $found[\{u, v\}] = true$; $\textsc{Add}(v, S)$ ;
```

1. Compare the bfs and dfs algorithms with the above code. Take special care in understanding visited.
2. Let us look at the sequence record[1],record[2],...,record[n]. Show that there is a path from record[i] to 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.

Exercise: organize field trip (Final 24)

Exercise 15.21

Let us suppose there are n kids in a class and they are planning a field trip. Some are friends and some are not. Their teacher asked each of them for their restrictions. The kids may declare their restrictions in the following four possible ways. (A kid may declare more than one restrictions)

- If X is going, then I will go. (friends)
- If X is going, then I will not go. (enemies)
- If X is not going, then I will go. (vengeful enemies)
- If X is not going, then I will not go. (great friends)

where X is some other kid.

1. Give an $O(n^2)$ algorithm to check if all restrictions of the kids can be satisfied.
2. Give an $O(n^2)$ algorithm to find a set of students that may go to the field trip without violating any of the restrictions.

Exercise: Topological sort via arrival times (Final 24)

Exercise 15.22

Consider the following modified version of TopologicalSort, where we are using arrival times instead of departure times.

Algorithm 15.15: TopologicalSortArrival(directed graph $G = (V, E)$)

```
DFSFull($G, v$);
if $\exists(v, v') \in E$ such that $v.arrival \geq v'.arrival$ then
    return "Cycle found: Sorting not possible";
return sorted vertices of $V$ in the increasing order of $arrival$.
```

Prove/Disprove that the above algorithm is correct, i.e., the algorithm returns a valid topological sort of the input directed graph.

Topic 15.10

Extra slides: Topological sort

Algorithm: Topological sort using DFS

An implementation with cycle detection and in-place sorting.

```
Algorithm 15.16: TopologicalSort( Directed graph $G = (V, E)$ )
1 Stack Sorted;
2 while $\exists v$ such that $v.visited == False$ do visit($v$) ;
```

```
Algorithm 15.17: 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 15