Lecture 13: Basic Graph Algorithms

Michael Dinitz

October 12, 2021
601.433/633 Introduction to Algorithms

Introduction

Next 3-4 weeks: graphs!

- Super important abstractions, used all over the place in CS
- Most of my research is in graph algorithms (particularly when graph represents computer/communication network)
- Great course on Graph Theory in AMS

Today: review of basic graph algorithms from Data Structures, one or two new

- Going to move pretty quickly, since much review: see CLRS for details!

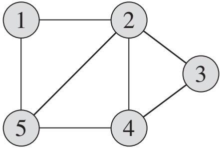
Basic Definitions

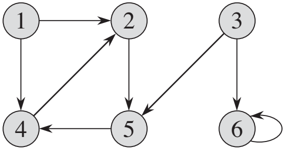
Definition

A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is a pair where $\mathbf{V}$ is a set and $\mathbf{E} \subseteq \binom{\mathbf{V}}{2}$ (unordered pairs) or $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ (ordered pairs).

Notation:

- Elements of $\mathbf{V}$ are called *vertices* or *nodes*
- Elements of $\mathbf{E}$ are called *edges* or *arcs*.
- If $\mathbf{E} \subseteq \binom{\mathbf{V}}{2}$ then graph is *undirected*, if $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ graph is *directed*
- $|\mathbf{V}| = \mathbf{n}$ and $|\mathbf{E}| = \mathbf{m}$ (usually)
- So "size of input" $= \mathbf{n} + \mathbf{m}$





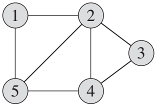
Representations

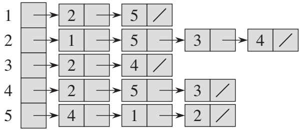
Adjacency List:

- Array **A** of length **n**
- **A[v]** is linked list of vertices *adjacent* to **v** (edge from **u** to **v**)

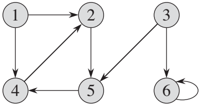
Adjacency Matrix:

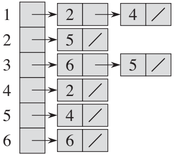
- $\mathbf{A} \in \{0,1\}^{n \times n}$
- $\mathbf{A_{ij}} = \begin{cases} \mathbf{1} & \text{if } (\mathbf{i},\mathbf{j}) \in \mathbf{E} \\ \mathbf{0} & \text{otherwise} \end{cases}$





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Representations (cont'd)

Adjacency List:

- Pros:
  - $\mathbf{O(n+m)}$ space
  - Can iterate through edges adjacent to $\mathbf{v}$ very efficiently
- Cons:
  - Hard to check of an edge exists: $\mathbf{O(d(u))}$ or $\mathbf{O(d(v))}$ (where $\mathbf{d}$ is the degree of $\mathbf{v}$: # edges with $\mathbf{v}$ as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $\mathbf{e = (u, v)}$ an edge in $\mathbf{O(1)}$ time
- Cons:
  - Takes $\mathbf{\Theta(n^2)}$ space: if $\mathbf{m}$ small, lots wasted!
  - Iterating through edges incident on $\mathbf{v}$ takes time $\mathbf{\Theta(n)}$, even if $\mathbf{d(v)}$ small.

Representations (cont'd)

Adjacency List:

- Pros:
  - $\mathbf{O(n + m)}$ space
  - Can iterate through edges adjacent to $\mathbf{v}$ very efficiently
- Cons:
  - Hard to check of an edge exists: $\mathbf{O(d(u))}$ or $\mathbf{O(d(v))}$ (where $\mathbf{d}$ is the degree of $\mathbf{v}$: # edges with $\mathbf{v}$ as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $\mathbf{e = (u, v)}$ an edge in $\mathbf{O(1)}$ time
- Cons:
  - Takes $\mathbf{\Theta(n^2)}$ space: if $\mathbf{m}$ small, lots wasted!
  - Iterating through edges incident on $\mathbf{v}$ takes time $\mathbf{\Theta(n)}$, even if $\mathbf{d(v)}$ small.

This class: adjacency list unless otherwise specified.

Representations (cont'd)

Adjacency List:

- Pros:
  - $\mathbf{O(n+m)}$ space
  - Can iterate through edges adjacent to $\mathbf{v}$ very efficiently
- Cons:
  - Hard to check of an edge exists: $\mathbf{O(d(u))}$ or $\mathbf{O(d(v))}$ (where $\mathbf{d}$ is the degree of $\mathbf{v}$: # edges with $\mathbf{v}$ as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $\mathbf{e = (u, v)}$ an edge in $\mathbf{O(1)}$ time
- Cons:
  - Takes $\mathbf{\Theta(n^2)}$ space: if $\mathbf{m}$ small, lots wasted!
  - Iterating through edges incident on $\mathbf{v}$ takes time $\mathbf{\Theta(n)}$, even if $\mathbf{d(v)}$ small.

This class: adjacency list unless otherwise specified.

Any way to improve these?

Representations (cont'd)

Adjacency List:

- Pros:
  - $\mathbf{O(n+m)}$ space
  - Can iterate through edges adjacent to $\mathbf{v}$ very efficiently
- Cons:
  - Hard to check of an edge exists: $\mathbf{O(d(u))}$ or $\mathbf{O(d(v))}$ (where $\mathbf{d}$ is the degree of $\mathbf{v}$: # edges with $\mathbf{v}$ as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $\mathbf{e = (u, v)}$ an edge in $\mathbf{O(1)}$ time
- Cons:
  - Takes $\mathbf{\Theta(n^2)}$ space: if $\mathbf{m}$ small, lots wasted!
  - Iterating through edges incident on $\mathbf{v}$ takes time $\mathbf{\Theta(n)}$, even if $\mathbf{d(v)}$ small.

This class: adjacency list unless otherwise specified.

Any way to improve these?

- Replace adjacency *list* with adjacency *structure*: Red-black tree, hash table, etc.
- Not traditional, doesn't gain us much, and more complicated. But better!

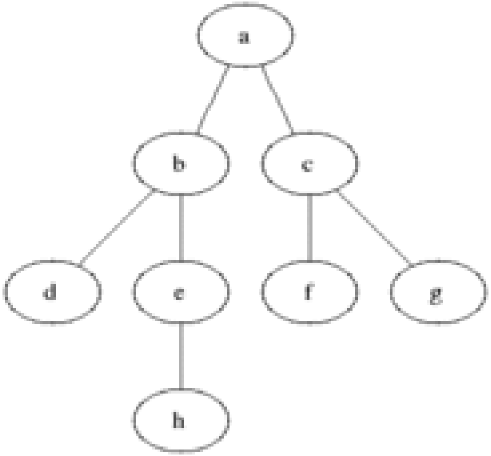
Breadth-First Search (BFS)

BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**

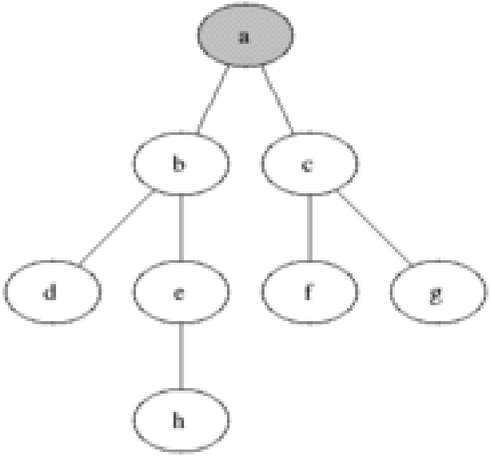
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



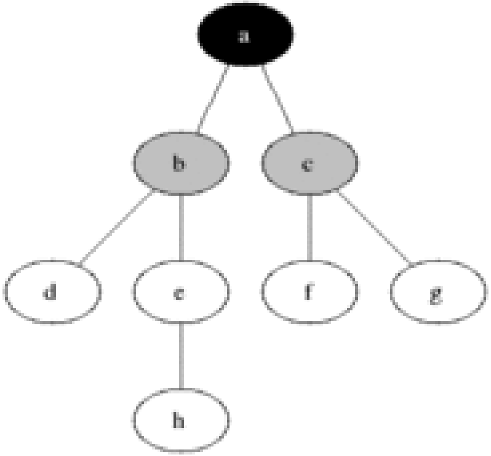
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



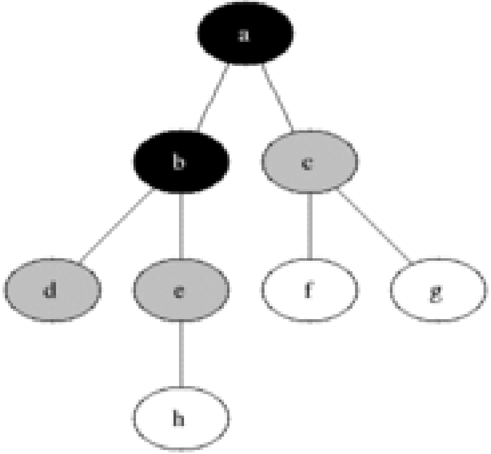
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



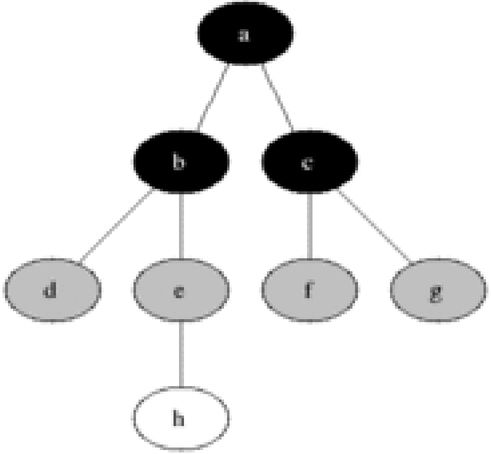
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



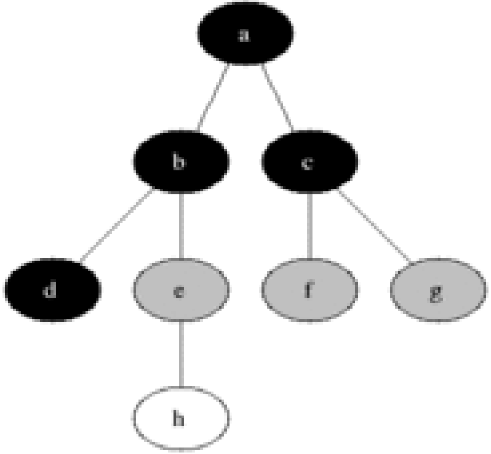
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



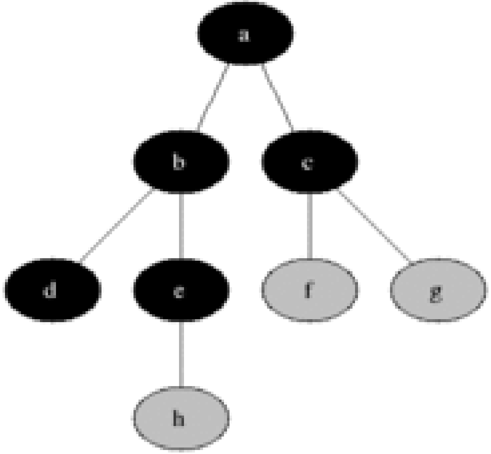
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



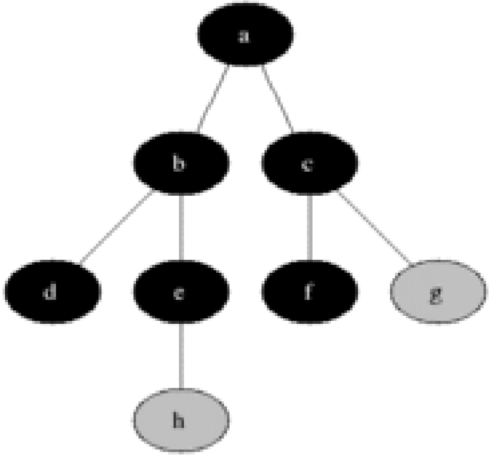
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



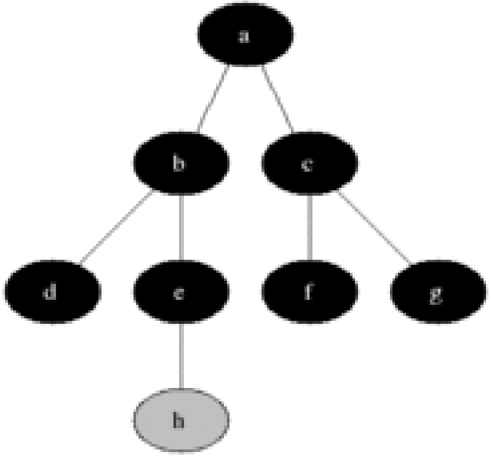
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



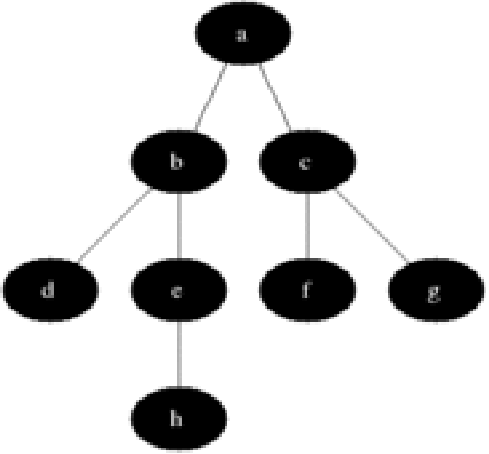
BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



BFS Definition

Idea: explore graph in *levels* or *layers* from source **s**



BFS Pseudocode

Idea: explore with a queue (LIFO)

```
BFS(G = (V, E), s) {
    Set mark(s) = True;
    Set mark(v) = False for all v ∈ V ∖ {s};
    Enqueue(s);
    while(queue not empty) {
        v = Dequeue();
        forall neighbors u of v {
            if(mark(u) == False) {
                mark(u) = True;
                Enqueue(u);
            }
        }
    }
}
```

BFS Pseudocode

Idea: explore with a queue (LIFO)

```
BFS(G = (V, E), s) {
    Set mark(s) = True;
    Set mark(v) = False for all v ∈ V ∖ {s};
    Enqueue(s);
    while(queue not empty) {
        v = Dequeue();
        forall neighbors u of v {
            if(mark(u) == False) {
                mark(u) = True;
                Enqueue(u);
            }
        }
    }
}
```

Running Time:

BFS Pseudocode

Idea: explore with a queue (LIFO)

```
BFS(G = (V, E), s) {
    Set mark(s) = True;
    Set mark(v) = False for all v ∈ V ∖ {s};
    Enqueue(s);
    while(queue not empty) {
        v = Dequeue();
        forall neighbors u of v {
            if(mark(u) == False) {
                mark(u) = True;
                Enqueue(u);
            }
        }
    }
}
```

Running Time: $O(n + m)$

BFS Pseudocode

Idea: explore with a queue (LIFO)

```
BFS(G = (V, E), s) {
    Set mark(s) = True;
    Set mark(v) = False for all v ∈ V ∖ {s};
    Enqueue(s);
    while(queue not empty) {
        v = Dequeue();
        forall neighbors u of v {
            if(mark(u) == False) {
                mark(u) = True;
                Enqueue(u);
            }
        }
    }
}
```

Running Time: $O(n + m)$

- $O(n)$ for initialization
- $O(m)$ for main while loop
  - Examine every edge twice: when each endpoint dequeued
  - Or (equivalent): Adjacency list scanned only when vertex dequeued

Correctness / Shortest Paths

Definition: Distance $\mathbf{d(u,v)}$ from **u** to **v** is min # edges in any path from **u** to **v**

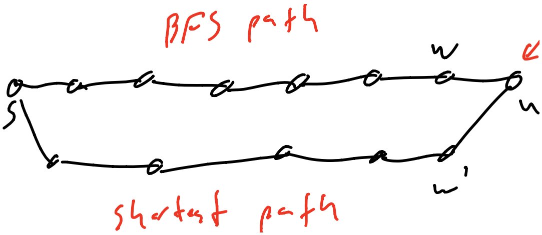
Theorem (informal): BFS(**s**) gives shortest paths from **s** to all other nodes

Correctness / Shortest Paths

Definition: Distance $\mathbf{d(u,v)}$ from **u** to **v** is min # edges in any path from **u** to **v**

Theorem (informal): BFS(**s**) gives shortest paths from **s** to all other nodes

Proof Sketch:
Assume false for contradiction, let **u** be closest node to **s** where BFS(**s**) doesn't give shortest path



$\mathbf{d(s,w') < d(s,w)}$

$\Longrightarrow$ **w′** dequeued before **w** (since **w′** has correct distance by def of **u**)

$\Longrightarrow$ **u** will be enqueued from **w′**, not **w**. Contradiction.

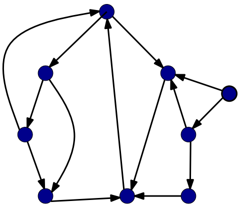
Depth-First Search (DFS)

DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

Init: for each $\mathbf{v} \in \mathbf{V}$, $\mathbf{mark(v)} = \mathbf{False}$;

DFS($\mathbf{v}$) {
    $\mathbf{mark(v)} = \mathbf{True}$;
    for each edge $\mathbf{(v, u)} \in \mathbf{A[v]}$ {
        if $\mathbf{mark(u)} == \mathbf{False}$ then DFS($\mathbf{u}$);
    }
}

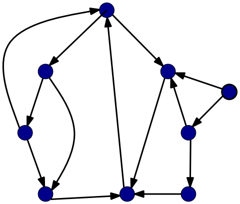


DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

```
Init: for each v ∈ V, mark(v) = False;

DFS(v) {
    mark(v) = True;
    for each edge (v, u) ∈ A[v] {
        if mark(u) == False then DFS(u);
    }
}
```



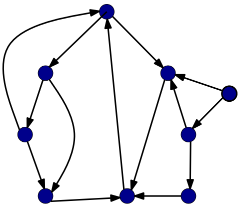
Running time:

DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

Init: for each $\mathbf{v} \in \mathbf{V}$, $\mathbf{mark(v)} = \mathbf{False}$;

DFS($\mathbf{v}$) {
  $\mathbf{mark(v)} = \mathbf{True}$;
  for each edge $\mathbf{(v, u)} \in \mathbf{A[v]}$ {
    if $\mathbf{mark(u)} == \mathbf{False}$ then DFS($\mathbf{u}$);
  }
}



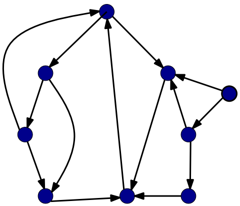
Running time: $\mathbf{O(m + n)}$

DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

Init: for each $\mathbf{v} \in \mathbf{V}$, $\mathbf{mark(v)} = \mathbf{False}$;

DFS($\mathbf{v}$) {
    $\mathbf{mark(v)} = \mathbf{True}$;
    for each edge $\mathbf{(v, u)} \in \mathbf{A[v]}$ {
        if $\mathbf{mark(u)} == \mathbf{False}$ then DFS($\mathbf{u}$);
    }
}



Running time: $\mathbf{O(m+n)}$

- $\mathbf{O(n)}$ initialization
- Every edge considered at most twice

DFS: Correctness

Definition: $\mathbf{u}$ is *reachable* from $\mathbf{v}$ if there is a path $\mathbf{v} = \mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_k} = \mathbf{u}$ such that $(\mathbf{v_i}, \mathbf{v_{i+1}}) \in \mathbf{E}$ for all $\mathbf{i} \in \{0, 1, \ldots, \mathbf{k} - 1\}$.

Theorem

When DFS($\mathbf{v}$) terminates, it has visited (marked) all nodes that are reachable from $\mathbf{v}$.

Proof.

Suppose $\mathbf{u}$ reachable from $\mathbf{v}$ but not marked when DFS($\mathbf{v}$) terminates.

DFS: Correctness

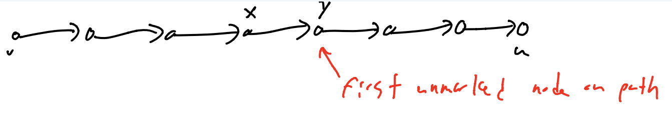
Definition: $\mathbf{u}$ is *reachable* from $\mathbf{v}$ if there is a path $\mathbf{v} = \mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_k} = \mathbf{u}$ such that $(\mathbf{v_i}, \mathbf{v_{i+1}}) \in \mathbf{E}$ for all $\mathbf{i} \in \{0, 1, \ldots, \mathbf{k} - 1\}$.

Theorem

When DFS($\mathbf{v}$) terminates, it has visited (marked) all nodes that are reachable from $\mathbf{v}$.

Proof.

Suppose $\mathbf{u}$ reachable from $\mathbf{v}$ but not marked when DFS($\mathbf{v}$) terminates.



DFS: Correctness

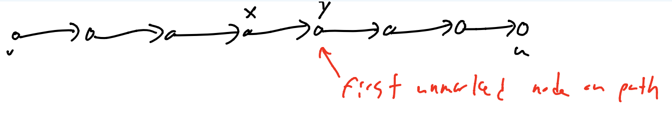
Definition: $\mathbf{u}$ is *reachable* from $\mathbf{v}$ if there is a path $\mathbf{v} = \mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_k} = \mathbf{u}$ such that $(\mathbf{v_i}, \mathbf{v_{i+1}}) \in \mathbf{E}$ for all $\mathbf{i} \in \{0, 1, \ldots, \mathbf{k} - 1\}$.

Theorem

When DFS($\mathbf{v}$) terminates, it has visited (marked) all nodes that are reachable from $\mathbf{v}$.

Proof.

Suppose $\mathbf{u}$ reachable from $\mathbf{v}$ but not marked when DFS($\mathbf{v}$) terminates.



$\mathbf{x}$ is marked so DFS($\mathbf{x}$) must have been called

DFS: Correctness

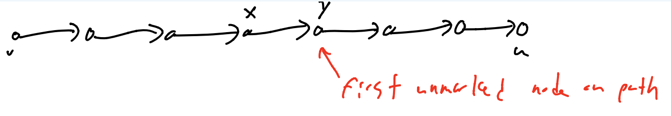
Definition: $\mathbf{u}$ is *reachable* from $\mathbf{v}$ if there is a path $\mathbf{v} = \mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_k} = \mathbf{u}$ such that $(\mathbf{v_i}, \mathbf{v_{i+1}}) \in \mathbf{E}$ for all $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{k} - \mathbf{1}\}$.

Theorem

When DFS($\mathbf{v}$) terminates, it has visited (marked) all nodes that are reachable from $\mathbf{v}$.

Proof.

Suppose $\mathbf{u}$ reachable from $\mathbf{v}$ but not marked when DFS($\mathbf{v}$) terminates.



$\mathbf{x}$ is marked so DFS($\mathbf{x}$) must have been called

$\implies$ $\mathbf{y}$ was either marked or DFS($\mathbf{y}$) called and it became marked.

DFS: Correctness

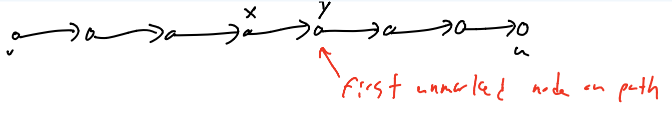
Definition: $\mathbf{u}$ is *reachable* from $\mathbf{v}$ if there is a path $\mathbf{v} = \mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_k} = \mathbf{u}$ such that $(\mathbf{v_i}, \mathbf{v_{i+1}}) \in \mathbf{E}$ for all $\mathbf{i} \in \{0, 1, \ldots, \mathbf{k} - 1\}$.

Theorem

When DFS($\mathbf{v}$) terminates, it has visited (marked) all nodes that are reachable from $\mathbf{v}$.

Proof.

Suppose $\mathbf{u}$ reachable from $\mathbf{v}$ but not marked when DFS($\mathbf{v}$) terminates.



$\mathbf{x}$ is marked so DFS($\mathbf{x}$) must have been called

$\Longrightarrow$ $\mathbf{y}$ was either marked or DFS($\mathbf{y}$) called and it became marked.

Contradiction. □

Graph variant

After DFS(**v**), node marked if and only if reachable from **v**.

Might want to continue until all nodes marked.

```
DFS(G) {
    for all v ∈ V, set mark(v) = False;
    while there exists an unmarked node v {
        DFS(v);
    }
}
```

Timestamps

Explicitly keep track of "start" and "finishing" times

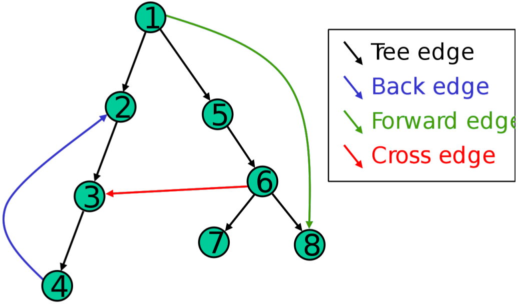
- Replaces **mark**

```
DFS(G) {
    t = 0;
    for all v ∈ V {
        start(v) = 0;
        finish(v) = 0;
    }
    while ∃v ∈ V with start(v) = 0 {
        DFS(v);
    }
}
```

```
DFS(v) {
    t = t + 1;
    start(v) = t;
    for each edge (v, u) ∈ A[v] {
        if start(u) == 0 then DFS(u);
    }
    t = t + 1;
    finish(v) = t;
}
```

Edge Types

DFS naturally gives a spanning forest: edge $(\mathbf{v}, \mathbf{u})$ if DFS($\mathbf{v}$) calls DFS($\mathbf{u}$)



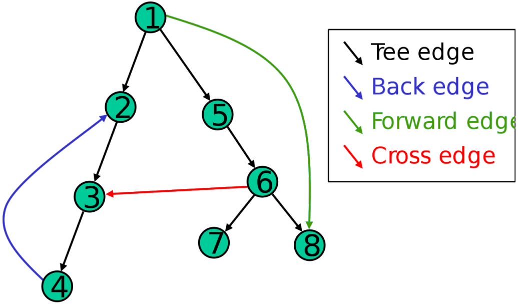
Forward Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ descendent of $\mathbf{v}$ (includes tree edges)

Back Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ an ancestor of $\mathbf{v}$

Cross Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ neither a descendent nor an ancestor of $\mathbf{v}$

Edge Types

DFS naturally gives a spanning forest: edge $(\mathbf{v}, \mathbf{u})$ if DFS($\mathbf{v}$) calls DFS($\mathbf{u}$)



Forward Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ descendent of $\mathbf{v}$ (includes tree edges)

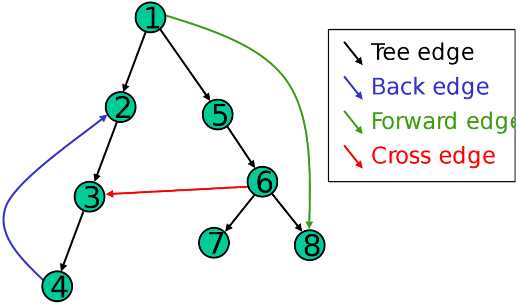
$$\mathbf{start}(\mathbf{v}) < \mathbf{start}(\mathbf{u}) < \mathbf{finish}(\mathbf{u}) < \mathbf{finish}(\mathbf{v})$$

Back Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ an ancestor of $\mathbf{v}$

Cross Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ neither a descendent nor an ancestor of $\mathbf{v}$

Edge Types

DFS naturally gives a spanning forest: edge $(\mathbf{v},\mathbf{u})$ if DFS($\mathbf{v}$) calls DFS($\mathbf{u}$)



Forward Edges: $(\mathbf{v},\mathbf{u})$ such that $\mathbf{u}$ descendent of $\mathbf{v}$ (includes tree edges)

$\mathbf{start(v) < start(u) < finish(u) < finish(v)}$

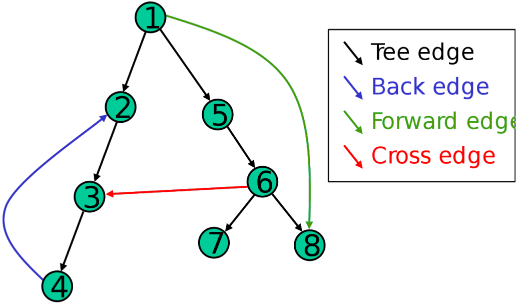
Back Edges: $(\mathbf{v},\mathbf{u})$ such that $\mathbf{u}$ an ancestor of $\mathbf{v}$

$\mathbf{start(u) < start(v) < finish(v) < finish(u)}$

Cross Edges: $(\mathbf{v},\mathbf{u})$ such that $\mathbf{u}$ neither a descendent nor an ancestor of $\mathbf{v}$

Edge Types

DFS naturally gives a spanning forest: edge $(\mathbf{v}, \mathbf{u})$ if DFS($\mathbf{v}$) calls DFS($\mathbf{u}$)



Forward Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ descendent of $\mathbf{v}$ (includes tree edges)

$\mathbf{start(v) < start(u) < finish(u) < finish(v)}$

Back Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ an ancestor of $\mathbf{v}$

$\mathbf{start(u) < start(v) < finish(v) < finish(u)}$

Cross Edges: $(\mathbf{v}, \mathbf{u})$ such that $\mathbf{u}$ neither a descendent nor an ancestor of $\mathbf{v}$

$\mathbf{start(u) < finish(u) < start(v) < finish(v)}$

Topological Sort

Definitions

Definition

A directed graph **G** is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.

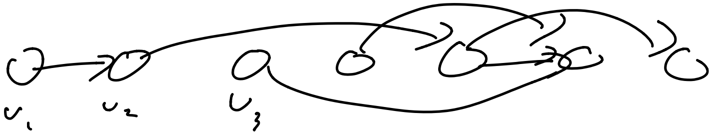
Definitions

Definition

A directed graph $\mathbf{G}$ is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.

Definition

A *topological sort* $\mathbf{v_1, v_2, \ldots, v_n}$ of a DAG is an ordering of the vertices such that all edges are of the form $(\mathbf{v_i}, \mathbf{v_j})$ with $\mathbf{i} < \mathbf{j}$.



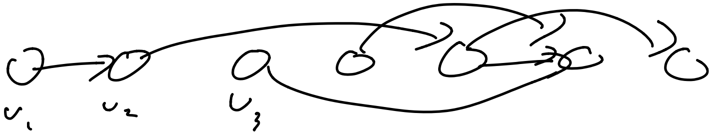
Definitions

Definition

A directed graph $\mathbf{G}$ is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.

Definition

A *topological sort* $\mathbf{v_1, v_2, \ldots, v_n}$ of a DAG is an ordering of the vertices such that all edges are of the form $(\mathbf{v_i}, \mathbf{v_j})$ with $\mathbf{i < j}$.



Can use DFS to characterize DAGs and compute topological sort!

Characterizing DAGs

Theorem

A directed graph $\mathbf{G}$ *is a DAG if and only if DFS*($\mathbf{G}$) *has no back edges.*

Characterizing DAGs

Theorem

A directed graph $\mathbf{G}$ *is a DAG if and only if DFS(*$\mathbf{G}$*) has no back edges.*

Proof.

Only if: contrapositive. If $\mathbf{G}$ has a back edge:

Characterizing DAGs

Theorem

A directed graph $\mathbf{G}$ *is a DAG if and only if DFS(*$\mathbf{G}$*) has no back edges.*

Proof.

Only if: contrapositive. If $\mathbf{G}$ has a back edge: Directed cycle! Not a DAG.

Characterizing DAGs

Theorem

A directed graph $\mathbf{G}$ *is a DAG if and only if DFS(*$\mathbf{G}$*) has no back edges.*

Proof.

Only if: contrapositive. If $\mathbf{G}$ has a back edge: Directed cycle! Not a DAG.

If: contrapositive. If $\mathbf{G}$ has a directed cycle $\mathbf{C}$:

Characterizing DAGs

Theorem

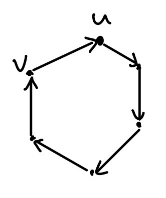
A directed graph $\mathbf{G}$ *is a DAG if and only if DFS(*$\mathbf{G}$*) has no back edges.*

Proof.

Only if: contrapositive. If $\mathbf{G}$ has a back edge: Directed cycle! Not a DAG.

If: contrapositive. If $\mathbf{G}$ has a directed cycle $\mathbf{C}$:

- Let $\mathbf{u} \in \mathbf{C}$ with minimum start value, $\mathbf{v}$ predecessor in cycle
- All nodes in $\mathbf{C}$ reachable from $\mathbf{u} \implies$ all nodes in $\mathbf{C}$ descendants of $\mathbf{u}$
- $(\mathbf{v}, \mathbf{u})$ a back edge



□

Topological Sort

- Run DFS($\mathbf{G}$)
  - When DFS($\mathbf{v}$) returns, put $\mathbf{v}$ at beginning of list

Topological Sort

- Run DFS($\mathbf{G}$)
  - When DFS($\mathbf{v}$) returns, put $\mathbf{v}$ at beginning of list

Correctness: Since $\mathbf{G}$ a DAG, never see back edge

$\Longrightarrow$ Every edge $(\mathbf{v}, \mathbf{u})$ out of $\mathbf{v}$ a forward or cross edge

$\Longrightarrow$ $\mathbf{finish(u)} < \mathbf{finish(v)}$

$\Longrightarrow$ $\mathbf{u}$ already in list

Topological Sort

- Run DFS($\mathbf{G}$)
  - When DFS($\mathbf{v}$) returns, put $\mathbf{v}$ at beginning of list

Correctness: Since $\mathbf{G}$ a DAG, never see back edge

$\implies$ Every edge $(\mathbf{v}, \mathbf{u})$ out of $\mathbf{v}$ a forward or cross edge

$\implies$ $\mathbf{finish(u) < finish(v)}$

$\implies$ $\mathbf{u}$ already in list

Running Time: $\mathbf{O(m + n)}$

Strongly Connected Components (SCC): Sketch

Definitions

Another application of DFS. "Kosaraju's Algorithm": Developed by Rao Kosaraju, professor emeritus at JHU CS!

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a directed graph.

Definition

$\mathbf{C} \subseteq \mathbf{V}$ is a *strongly connected component (SCC)* if it is a *maximal* subset such that for all $\mathbf{u}, \mathbf{v} \in \mathbf{C}$, $\mathbf{u}$ can reach $\mathbf{v}$ and vice versa.

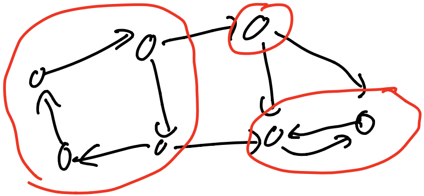
Definitions

Another application of DFS. "Kosaraju's Algorithm": Developed by Rao Kosaraju, professor emeritus at JHU CS!

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a directed graph.

Definition

$\mathbf{C} \subseteq \mathbf{V}$ is a *strongly connected component (SCC)* if it is a *maximal* subset such that for all $\mathbf{u}, \mathbf{v} \in \mathbf{C}$, $\mathbf{u}$ can reach $\mathbf{v}$ and vice versa.



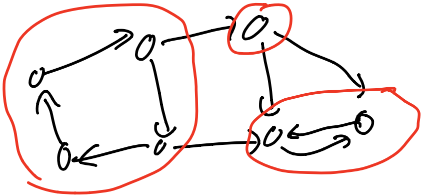
Definitions

Another application of DFS. "Kosaraju's Algorithm": Developed by Rao Kosaraju, professor emeritus at JHU CS!

$\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a directed graph.

Definition

$\mathbf{C} \subseteq \mathbf{V}$ is a *strongly connected component (SCC)* if it is a *maximal* subset such that for all $\mathbf{u}, \mathbf{v} \in \mathbf{C}$, $\mathbf{u}$ can reach $\mathbf{v}$ and vice versa.



Fact: There is a *unique* partition of $\mathbf{V}$ into SCCs

Proof: Bireachability is an equivalence relation

SCC Problem

Problem: Give $\mathbf{G}$ compute SCCs (partition $\mathbf{V}$ into the SCCs)

SCC Problem

Problem: Give **G** compute SCCs (partition **V** into the SCCs)

Trivial Algorithm:

SCC Problem

Problem: Give $\mathbf{G}$ compute SCCs (partition $\mathbf{V}$ into the SCCs)

Trivial Algorithm: DFS/BFS from every node, keep track of what's reachable from where

SCC Problem

Problem: Give $\mathbf{G}$ compute SCCs (partition $\mathbf{V}$ into the SCCs)

Trivial Algorithm: DFS/BFS from every node, keep track of what's reachable from where

- Running time: $\mathbf{O(n(m+n))}$

SCC Problem

Problem: Give $\mathbf{G}$ compute SCCs (partition $\mathbf{V}$ into the SCCs)

Trivial Algorithm: DFS/BFS from every node, keep track of what's reachable from where

- Running time: $\mathbf{O(n(m+n))}$

Can we do better? $\mathbf{O(m+n)}$?

Graph of SCCs

Definition: Let $\hat{\mathbf{G}}$ be graph of SCCs:

- Vertex $\mathbf{v(C)}$ for each SCC $\mathbf{C}$
- Edge $\mathbf{(v(C), v(C'))}$ if $\exists\ \mathbf{u} \in \mathbf{C}, \mathbf{v} \in \mathbf{C'}$ such that $\mathbf{(u, v)} \in \mathbf{E}$

Graph of SCCs

Definition: Let $\hat{\mathbf{G}}$ be graph of SCCs:

- Vertex $\mathbf{v(C)}$ for each SCC $\mathbf{C}$
- Edge $\mathbf{(v(C), v(C'))}$ if $\exists\ \mathbf{u} \in \mathbf{C}, \mathbf{v} \in \mathbf{C'}$ such that $\mathbf{(u, v)} \in \mathbf{E}$

Theorem

$\hat{\mathbf{G}}$ *is a DAG.*

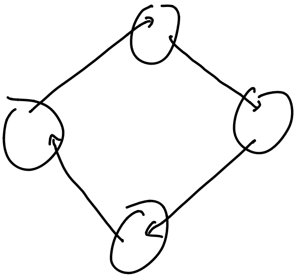
Graph of SCCs

Definition: Let $\hat{\mathbf{G}}$ be graph of SCCs:

- Vertex $\mathbf{v(C)}$ for each SCC $\mathbf{C}$
- Edge $\mathbf{(v(C), v(C'))}$ if $\exists\ \mathbf{u} \in \mathbf{C}, \mathbf{v} \in \mathbf{C'}$ such that $\mathbf{(u, v)} \in \mathbf{E}$

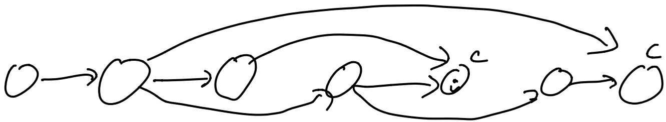
Theorem

$\hat{\mathbf{G}}$ *is a DAG.*



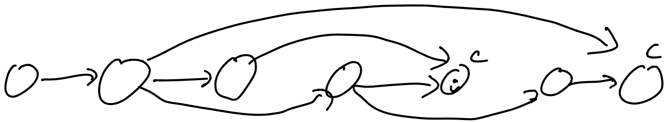
Sink SCC

Since $\hat{\mathbf{G}}$ a DAG, has a topological sort



Sink SCC

Since $\hat{\mathbf{G}}$ a DAG, has a topological sort

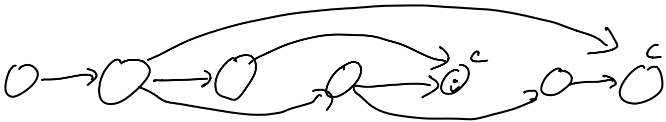


Definition: SCC **C** is a *sink* SCC if no outgoing edges

- At least one sink SCC exists

Sink SCC

Since $\hat{\mathbf{G}}$ a DAG, has a topological sort



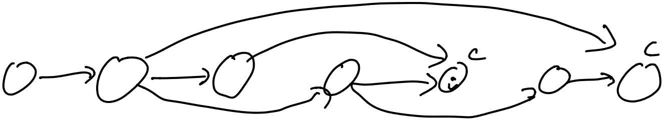
Definition: SCC **C** is a *sink* SCC if no outgoing edges

- At least one sink SCC exists

What happens if we run DFS($\mathbf{v}$) where $\mathbf{v}$ in a sink SCC?

Sink SCC

Since $\hat{\mathbf{G}}$ a DAG, has a topological sort



Definition: SCC **C** is a *sink* SCC if no outgoing edges

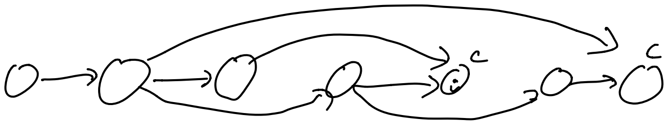
- At least one sink SCC exists

What happens if we run DFS(**v**) where **v** in a sink SCC?

- See exactly nodes in **C**!

Sink SCC

Since $\hat{\mathbf{G}}$ a DAG, has a topological sort



Definition: SCC $\mathbf{C}$ is a *sink* SCC if no outgoing edges

- At least one sink SCC exists

What happens if we run DFS($\mathbf{v}$) where $\mathbf{v}$ in a sink SCC?

- See exactly nodes in $\mathbf{C}$!

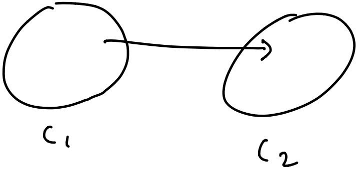
Strategy: find node in sink SCC, run DFS, remove nodes found, repeat

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C)} = \mathbf{max}_{\mathbf{v} \in \mathbf{C}}\, \mathbf{finish(v)}$

Lemma

Let $\mathbf{C_1, C_2}$ *distinct SCCs s.t.* $(\mathbf{v(C_1), v(C_2)}) \in \mathbf{E(\hat{G})}$. *Then* $\mathbf{finish(C_1) > finish(C_2)}$.



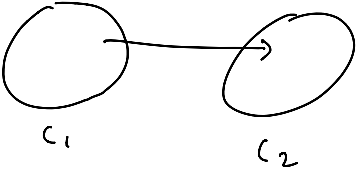
Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C) = max_{v \in C}\, finish(v)}$

Lemma

Let $\mathbf{C_1, C_2}$ *distinct SCCs s.t.* $\mathbf{(v(C_1), v(C_2)) \in E(\hat{G})}$*. Then* $\mathbf{finish(C_1) > finish(C_2)}$*.*



Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

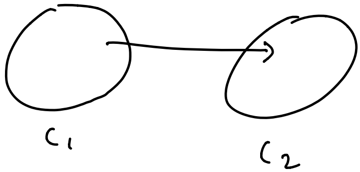
- If $\mathbf{x} \in \mathbf{C_1}$:

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C) = max_{v \in C}\, finish(v)}$

Lemma

Let $\mathbf{C_1, C_2}$ *distinct SCCs s.t.* $\mathbf{(v(C_1), v(C_2)) \in E(\hat{G})}$. *Then* $\mathbf{finish(C_1) > finish(C_2)}$.



Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

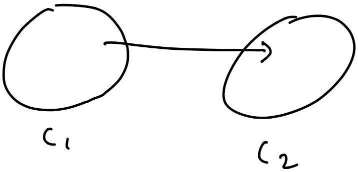
- If $\mathbf{x} \in \mathbf{C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C)} = \mathbf{max}_{\mathbf{v} \in \mathbf{C}}\, \mathbf{finish(v)}$

Lemma

Let $\mathbf{C_1}, \mathbf{C_2}$ *distinct SCCs s.t.* $(\mathbf{v(C_1)}, \mathbf{v(C_2)}) \in \mathbf{E(\hat{G})}$. *Then* $\mathbf{finish(C_1)} > \mathbf{finish(C_2)}$.



Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

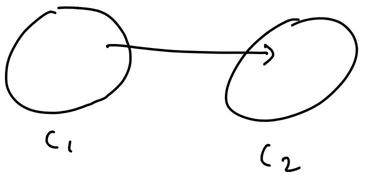
- If $\mathbf{x} \in \mathbf{C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished
- If $\mathbf{x} \in \mathbf{C_2}$:

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C)} = \mathbf{max}_{\mathbf{v} \in \mathbf{C}}\, \mathbf{finish(v)}$

Lemma

Let $\mathbf{C_1}, \mathbf{C_2}$ *distinct SCCs s.t.* $(\mathbf{v(C_1)}, \mathbf{v(C_2)}) \in \mathbf{E}(\hat{\mathbf{G}})$. *Then* $\mathbf{finish(C_1)} > \mathbf{finish(C_2)}$.



Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

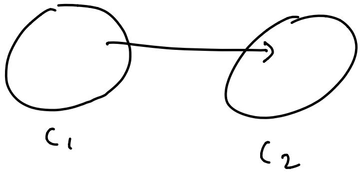
- If $\mathbf{x} \in \mathbf{C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished
- If $\mathbf{x} \in \mathbf{C_2}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$ but nothing from $\mathbf{C_1}$, so $\mathbf{x}$ finishes before any node in $\mathbf{C_1}$ starts

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C)} = \mathbf{max}_{\mathbf{v} \in \mathbf{C}}\, \mathbf{finish(v)}$

Lemma

Let $\mathbf{C_1}, \mathbf{C_2}$ *distinct SCCs s.t.* $(\mathbf{v(C_1)}, \mathbf{v(C_2)}) \in \mathbf{E(\hat{G})}$. *Then* $\mathbf{finish(C_1)} > \mathbf{finish(C_2)}$.



Let $\mathbf{x} \in \mathbf{C_1} \cup \mathbf{C_2}$ be first node encountered by DFS

- If $\mathbf{x} \in \mathbf{C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished
- If $\mathbf{x} \in \mathbf{C_2}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$ but nothing from $\mathbf{C_1}$, so $\mathbf{x}$ finishes before any node in $\mathbf{C_1}$ starts

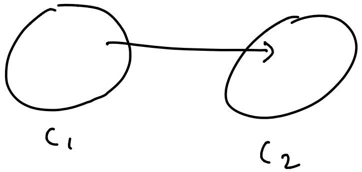
So node with max finish time in a *source* SCC. Want sink.

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C) = max_{v \in C} finish(v)}$

Lemma

Let $\mathbf{C_1, C_2}$ *distinct SCCs s.t.* $\mathbf{(v(C_1), v(C_2)) \in E(\hat{G})}$. *Then* $\mathbf{finish(C_1) > finish(C_2)}$.



Let $\mathbf{x \in C_1 \cup C_2}$ be first node encountered by DFS

- If $\mathbf{x \in C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished
- If $\mathbf{x \in C_2}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$ but nothing from $\mathbf{C_1}$, so $\mathbf{x}$ finishes before any node in $\mathbf{C_1}$ starts

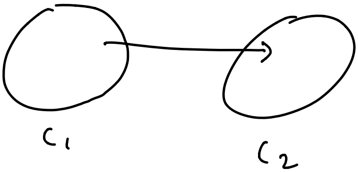
So node with max finish time in a *source* SCC. Want sink. Reverse all edges!

SCCs and DFS

Run DFS($\mathbf{G}$), and let $\mathbf{finish(C) = max_{v \in C}\, finish(v)}$

Lemma

Let $\mathbf{C_1, C_2}$ *distinct SCCs s.t.* $\mathbf{(v(C_1), v(C_2)) \in E(\hat{G})}$. *Then* $\mathbf{finish(C_1) > finish(C_2)}$.



Let $\mathbf{x \in C_1 \cup C_2}$ be first node encountered by DFS

- If $\mathbf{x \in C_1}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$, so DFS($\mathbf{x}$) does not complete until all of $\mathbf{C_2}$ finished
- If $\mathbf{x \in C_2}$: all of $\mathbf{C_2}$ reachable from $\mathbf{x}$ but nothing from $\mathbf{C_1}$, so $\mathbf{x}$ finishes before any node in $\mathbf{C_1}$ starts

So node with max finish time in a *source* SCC. Want sink. Reverse all edges!

Kosaraju's Algorithm

Definition: $\mathbf{G^T}$ is $\mathbf{G}$ with all edges reversed.

DFS($\mathbf{G^T}$) to get finishing times
while($\mathbf{G}$ non-empty) {
    Let $\mathbf{v}$ be vertex in $\mathbf{G}$ with largest finishing time (from original DFS of $\mathbf{G^T}$)
    Run DFS($\mathbf{v}$), let $\mathbf{C}$ be all nodes found
    Delete $\mathbf{C}$ from $\mathbf{G}$ as an SCC
}

Kosaraju's Algorithm

Definition: $\mathbf{G^T}$ is $\mathbf{G}$ with all edges reversed.

DFS($\mathbf{G^T}$) to get finishing times
while($\mathbf{G}$ non-empty) {
    Let $\mathbf{v}$ be vertex in $\mathbf{G}$ with largest finishing time (from original DFS of $\mathbf{G^T}$)
    Run DFS($\mathbf{v}$), let $\mathbf{C}$ be all nodes found
    Delete $\mathbf{C}$ from $\mathbf{G}$ as an SCC
}

Some implementation details missing (repeatedly finding max finishing time without using heap): see book

Kosaraju's Algorithm

Definition: $\mathbf{G^T}$ is $\mathbf{G}$ with all edges reversed.

DFS($\mathbf{G^T}$) to get finishing times
while($\mathbf{G}$ non-empty) {
    Let $\mathbf{v}$ be vertex in $\mathbf{G}$ with largest finishing time (from original DFS of $\mathbf{G^T}$)
    Run DFS($\mathbf{v}$), let $\mathbf{C}$ be all nodes found
    Delete $\mathbf{C}$ from $\mathbf{G}$ as an SCC
}

Some implementation details missing (repeatedly finding max finishing time without using heap): see book

Running Time: $\mathbf{O(m+n)}$

Correctness Sketch

Let $\mathbf{C_1, C_2, \ldots, C_k}$ be set identified by algorithm (in order)

Theorem

$\mathbf{C_i}$ *is a sink SCC of* $\mathbf{G} \setminus \left(\bigcup_{\mathbf{j=1}}^{\mathbf{i-1}} \mathbf{C_j} \right)$

Correctness Sketch

Let $\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_k}$ be set identified by algorithm (in order)

Theorem

$\mathbf{C_i}$ *is a sink SCC of* $\mathbf{G} \setminus \left(\bigcup_{j=1}^{i-1} \mathbf{C_j} \right)$

Induction on $\mathbf{i}$.

Correctness Sketch

Let $\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_k}$ be set identified by algorithm (in order)

Theorem

$\mathbf{C_i}$ *is a sink SCC of* $\mathbf{G} \setminus \left(\bigcup_{\mathbf{j=1}}^{\mathbf{i-1}} \mathbf{C_j}\right)$

Induction on $\mathbf{i}$.

Base case: $\mathbf{i} = \mathbf{1}$. By previous argument, largest finishing time in $\mathbf{G^T} \implies$ in sink SCC of $\mathbf{G}$ $\implies \mathbf{C_1}$ is sink SCC of $\mathbf{G}$

Correctness Sketch

Let $\mathbf{C_1, C_2, \dots, C_k}$ be set identified by algorithm (in order)

Theorem

$\mathbf{C_i}$ *is a sink SCC of* $\mathbf{G} \setminus \left(\bigcup_{\mathbf{j=1}}^{\mathbf{i-1}} \mathbf{C_j}\right)$

Induction on $\mathbf{i}$.

Base case: $\mathbf{i = 1}$. By previous argument, largest finishing time in $\mathbf{G^T} \implies$ in sink SCC of $\mathbf{G}$ $\implies \mathbf{C_1}$ is sink SCC of $\mathbf{G}$

Inductive case: Let $\mathbf{v}$ node remaining with largest finishing time.

- By induction, current graph is $\mathbf{G}$ minus $\mathbf{i - 1}$ SCCs of $\mathbf{G}$
- Implies $\mathbf{v}$ must be in sink SCC of remaining graph, so get an SCC of remaining graph when run DFS
- By induction, also an SCC of original graph