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

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

Notation:

- Elements of \( V \) are called vertices or nodes.
- Elements of \( E \) are called edges or arcs.
- If \( E \subseteq \binom{V}{2} \) then graph is undirected, if \( E \subseteq V \times V \) graph is directed.
- \(|V| = n \) and \(|E| = m \) (usually).
- So “size of input” = \( n + m \).
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:
- \( A \in \{0, 1\}^{n \times n} \)
- \( A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \)
Representations (cont’d)

Adjacency List:

- **Pros:**
  - $O(n + m)$ space
  - Can iterate through edges adjacent to $v$ very efficiently

- **Cons:**
  - Hard to check if an edge exists: $O(d(u))$ or $O(d(v))$ (where $d$ is the degree of $v$: # edges with $v$ as endpoint)

Adjacency Matrix:

- **Pros:**
  - Can check if $e = (u, v)$ an edge in $O(1)$ time

- **Cons:**
  - Takes $\Theta(n^2)$ space: if $m$ small, lots wasted!
  - Iterating through edges incident on $v$ takes time $\Theta(n)$, even if $d(v)$ 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!
Representations (cont’d)

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

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

This class: adjacency list unless otherwise specified.
Representations (cont’d)

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

Adjacency Matrix:
  ▶ Pros:
    ▶ Can check if $e = (u, v)$ an edge in $O(1)$ time
  ▶ Cons:
    ▶ Takes $\Theta(n^2)$ space: if $m$ small, lots wasted!
    ▶ Iterating through edges incident on $v$ takes time $\Theta(n)$, even if $d(v)$ 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!
Representations (cont’d)

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

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

**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
BFS Pseudocode

Idea: explore with a queue (LIFO)

\[
\text{BFS}(G = (V, E), s) \{ \\
\quad \text{Set } \text{mark}(s) = \text{True}; \\
\quad \text{Set } \text{mark}(v) = \text{False} \text{ for all } v \in V \setminus \{s\}; \\
\quad \text{Enqueue}(s); \\
\quad \text{while}(\text{queue not empty}) \{ \\
\quad \quad v = \text{Dequeue}(); \\
\quad \quad \text{forall neighbors } u \text{ of } v \{ \\
\quad \quad \quad \text{if}(\text{mark}(u) == \text{False}) \{ \\
\quad \quad \quad \quad \text{mark}(u) = \text{True}; \\
\quad \quad \quad \quad \text{Enqueue}(u); \\
\quad \quad \quad \} \\
\quad \quad \} \\
\quad \} \\
\}
\]

Running Time:

\[O(n + m) \leq O(n) \text{ for initialization} \leq O(m) \text{ for main while loop}\]

Examine every edge twice:

when each endpoint dequeued

Or (equivalent): Adjacency list scanned only when vertex dequeued
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 $v \in V \setminus \{s\}$;
    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 \( d(u, v) \) 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
Definition: Distance $d(u, v)$ 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

\[
\begin{align*}
    d(s, w') &< d(s, w) \\
    &\implies w' \text{ dequeued before } w \text{ (since } w' \text{ has correct distance by def of } u) \\
    &\implies u \text{ will be enqueued from } w', \text{ not } w. \text{ Contradiction.}
\end{align*}
\]
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 \( v \in V \), \( \text{mark}(v) = \text{False} \);

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

Basically, we look at each arc and if the other side has not already been visited yet, we recursively visit it. Here’s an example. The labeled nodes are the ones visited by calling \( \text{DFS}(A) \). The dashed edges are the ones not traversed, the dotted ones were not even looked at.

An node \( w \) is reachable from \( v \) in \( G \) if there is a path \( v = v_0, v_1, v_2, ..., v_k = w \) such that each \( (v_i, v_{i+1}) \) is an arc of \( G \).

Fact 1

When \( \text{DFS}(v) \) terminates, it has visited (marked) all the nodes that can be reached from \( v \).

Proof:
The simple proof is by induction. We will terminate because every call to \( \text{DFS}(v) \) is to an unmarked node, and each such call marks a node. There are \( n \) nodes, hence \( n \) calls, before we stop.

Now suppose some node \( w \) that is reachable from \( v \) and is not marked when \( \text{DFS}(v) \) terminates.

Since \( w \) is reachable, there is a path \( v = v_0, v_1, v_2, ..., v_k = w \) from \( v \) to \( w \), and a first node \( v_i \) on this path that is not marked. But this is impossible, because we marked \( v_i \) and would have examined the arc \( (v_i, v_{i+1}) \).

Of course, it may be the case that not all the nodes in \( G \) are reachable from \( v \). So really we should do the following:

\[
\text{DFS-graph(graph } G) \\
\text{for all } v \text{ in } V, \text{mark}(v) = \text{F}. \\
\text{While there exists an unmarked node } v \text{ } \text{DFS}(v)
\]

This process will visit all the nodes of the graph (just by the definition of the procedure). Here’s the old example.

It will help to have a few more pieces of data defined, which will make reasoning about DFS much easier. One is \( \text{active}(v) \), which is a flag that indicates that \( v \) is currently on the recursion stack.

Two other numbers are \( \text{pre}(v) \) and \( \text{post}(v) \) which are “times” at which we add \( v \) to the recursion stack, and when we remove \( v \) from it. (In 15-210, these were the times at which you enter \( v \) and exit \( v \).)

Here is the depth first search procedure:

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

Running time:

\( O(m + n) / \Omega(n) \)

initialization

Every edge considered at most \( 2 \) times.

Michael Dinitz
Lecture 13: Basic Graph Algorithms
October 12, 2021 11 / 27
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 \( v \in V \), \( \text{mark}(v) = \text{False} \);

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

Running time: \( O(m+n) \)
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 \( v \in V \), \( \text{mark}(v) = \text{False} \);

\[
\text{DFS}(v) \left\{ \\
\text{mark}(v) = \text{True} ; \\
\text{for each edge } (v, u) \in A[v] \left\{ \\
\text{if mark}(u) == \text{False} \text{ then } \text{DFS}(u) ; \\
\text{\}} \right. \\
\text{\}}
\]

Running time: \( O(m + n) \)
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 \( v \in V \), \( \text{mark}(v) = \text{False} \);

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

Running time: \( O(m + n) \)
- \( O(n) \) initialization
- Every edge considered at most twice
DFS: Correctness

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

**Theorem**

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

**Proof.**

Suppose \( u \) reachable from \( v \) but not marked when \( \text{DFS}(v) \) terminates.
DFS: Correctness

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

Theorem

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

Proof.

Suppose \( u \) reachable from \( v \) but not marked when DFS\((v)\) terminates.
**DFS: Correctness**

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

**Theorem**

*When DFS*(\( v \)) *terminates, it has visited (marked) all nodes that are reachable from* \( v \).*

**Proof.**

Suppose \( u \) reachable from \( v \) but not marked when DFS(\( v \)) terminates.

\[ x \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow y \]

\( x \) is marked so DFS(\( x \)) must have been called
DFS: Correctness

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

**Theorem**

*When DFS*(\( v \)) *terminates, it has visited (marked) all nodes that are reachable from* \( v \).*

**Proof.**

Suppose \( u \) reachable from \( v \) but not marked when DFS(\( v \)) terminates.

\[ \xymatrix{ & o \ar[rr] & & x \ar[r] & y \ar[r] & o \ar[r] & u \} \]

\( x \) is marked so DFS(\( x \)) must have been called

\[ \Rightarrow \quad y \text{ was either marked or DFS}(y) \text{ called and it became marked.} \]
DFS: Correctness

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

**Theorem**

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

**Proof.**

Suppose \( u \) reachable from \( v \) but not marked when DFS\((v)\) terminates.

\[ \text{x is marked so DFS}(x) \text{ must have been called} \]

\[ \Rightarrow \ y \text{ was either marked or DFS}(y) \text{ called and it became marked.} \]

Contradiction.
Graph variant

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

Might want to continue until all nodes marked.

```python
DFS(G) {
    for all $v \in V$, set $\text{mark}(v) = \text{False}$;
    while there exists an unmarked node $v$ {
        DFS($v$);
    }
}
```
**Timestamps**

Explicitly keep track of “start” and “finishing” times

- Replaces *mark*

---

**DFS(G) {**

- \( t = 0; \)
- \( \text{for all } v \in V \{ \]
  - \( \text{start}(v) = 0; \)
  - \( \text{finish}(v) = 0; \)
- \( \}
- \( \text{while } \exists v \in V \text{ with } \text{start}(v) = 0 \{ \]
  - \( \text{DFS}(v); \)
- \( \}

---

**DFS(v) {**

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

---
Edge Types

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

**Forward Edges:** \((v, u)\) such that \(u\) descendent of \(v\) (includes tree edges)

**Back Edges:** \((v, u)\) such that \(u\) an ancestor of \(v\)

**Cross Edges:** \((v, u)\) such that \(u\) neither a descendent nor an ancestor of \(v\)
Edge Types

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

**Forward Edges:** \((v, u)\) such that \(u\) descendent of \(v\) (includes tree edges)

\[\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v)\]

**Back Edges:** \((v, u)\) such that \(u\) an ancestor of \(v\)

**Cross Edges:** \((v, u)\) such that \(u\) neither a descendent nor an ancestor of \(v\)
Edge Types

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

**Forward Edges:** \((v, u)\) such that \(u\) descendent of \(v\) (includes tree edges)
\[
\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v)
\]

**Back Edges:** \((v, u)\) such that \(u\) an ancestor of \(v\)
\[
\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u)
\]

**Cross Edges:** \((v, u)\) such that \(u\) neither a descendent nor an ancestor of \(v\)
Edge Types

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

**Forward Edges:** \((v, u)\) such that \(u\) descendent of \(v\) (includes tree edges)

\[
\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v)
\]

**Back Edges:** \((v, u)\) such that \(u\) an ancestor of \(v\)

\[
\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u)
\]

**Cross Edges:** \((v, u)\) such that \(u\) neither a descendent nor an ancestor of \(v\)

\[
\text{start}(u) < \text{finish}(u) < \text{start}(v) < \text{finish}(v)
\]
Topological Sort
Definitions

Definition

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

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

Can use DFS to characterize DAGs and compute topological sort!
Definitions

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

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

Can use DFS to characterize DAGs and compute topological sort!
Theorem

A directed graph $G$ is a DAG if and only if $DFS(G)$ has no back edges.
Theorem

A directed graph $G$ is a DAG if and only if $DFS(G)$ has no back edges.

Proof.

Only if: contrapositive. If $G$ has a back edge:

1. Let $u \in C$ with minimum start value, $v$ predecessor in cycle.
2. All nodes in $C$ reachable from $u$ ($\Rightarrow$ all nodes in $C$ descendants of $u$).
3. $(v, u)$ a back edge.
Characterizing DAGs

**Theorem**

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

**Proof.**

Only if: contrapositive. If $G$ has a back edge: Directed cycle! Not a DAG.
Theorem

A directed graph $G$ is a DAG if and only if $DFS(G)$ has no back edges.

Proof.

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

If: contrapositive. If $G$ has a directed cycle $C$: 
Characterizing DAGs

Theorem

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

Proof.

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

If: contrapositive. If $G$ has a directed cycle $C$:

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

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

Correctness:
Since $G$ a DAG, never see back edge
\[ \text{⇒} \]
Every edge $(v, u)$ out of $v$ a forward or cross edge
\[ \text{⇒} \]
finish $(u) < \text{finish}(v)$
\[ \text{⇒} \]
u already in list

Running Time: $O(m + n)$
Topological Sort

- Run DFS(G)
  - When DFS(v) returns, put v at beginning of list

**Correctness:** Since G a DAG, never see back edge

\[\Rightarrow\] Every edge \((v, u)\) out of \(v\) a forward or cross edge

\[\Rightarrow\] \(\text{finish}(u) < \text{finish}(v)\)

\[\Rightarrow\] \(u\) already in list
Topological Sort

- Run DFS(G)
  - When DFS(v) returns, put v at beginning of list

**Correctness:** Since G a DAG, never see back edge

⇒ Every edge (v, u) out of v a forward or cross edge

⇒ finish(u) < finish(v)

⇒ u already in list

**Running Time:** $O(m + n)$
Strongly Connected Components (SCC): Sketch
Definitions

Another application of DFS. “Kosaraju’s Algorithm”: Developed by Rao Kosaraju, professor emeritus at JHU CS!

\[ G = (V, E) \] a directed graph.

**Definition**

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

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

\[ G = (V, E) \] a directed graph.

**Definition**

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

![Diagram of a directed graph with strongly connected components highlighted in red.](image-url)
Definitions

Another application of DFS. “Kosaraju’s Algorithm”: Developed by Rao Kosaraju, professor emeritus at JHU CS!

\[ G = (V, E) \] a directed graph.

**Definition**

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

**Fact:** There is a unique partition of \( V \) into SCCs

**Proof:** Bireachability is an equivalence relation
Problem: Give $G$ compute SCCs (partition $V$ into the SCCs)
**Problem:** Give $G$ compute SCCs (partition $V$ into the SCCs)

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

**Trivial Algorithm**: DFS/BFS from every node, keep track of what’s reachable from where
SCC Problem

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

**Trivial Algorithm:** DFS/BFS from every node, keep track of what’s reachable from where
- Running time: $O(n(m + n))$
Problem: Give $G$ compute SCCs (partition $V$ into the SCCs)

Trivial Algorithm: DFS/BFS from every node, keep track of what’s reachable from where
  - Running time: $O(n(m + n))$

Can we do better? $O(m + n)$?
Graph of SCCs

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

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

Theorem $\hat{G}$ is a DAG.
Graph of SCCs

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

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

**Theorem**

$\hat{G}$ is a DAG.
**Graph of SCCs**

**Definition:** Let $\hat{G}$ be graph of SCCs:
- Vertex $v(C)$ for each SCC $C$
- Edge $(v(C), v(C'))$ if $\exists u \in C, v \in C'$ such that $(u, v) \in E$

**Theorem**

$\hat{G}$ is a DAG.
Sink SCC

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

Since $\hat{G}$ 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 $\hat{G}$ 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?
Sink SCC

Since $\hat{G}$ 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 $\text{DFS}(v)$ where $v$ in a sink SCC?
- See exactly nodes in $C$!
Sink SCC

Since $\hat{G}$ 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 $\text{DFS}(v)$ where $v$ in a sink SCC?
  - See exactly nodes in $C$!

Strategy: find node in sink SCC, run DFS, remove nodes found, repeat
Run $\text{DFS}(G)$, and let $\text{finish}(C) = \max_{v \in C} \text{finish}(v)$

**Lemma**

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

Let $x \in C_1 \cup C_2$ be first node encountered by DFS.
SCCs and DFS

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

Lemma

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

Let \(x \in C_1 \cup C_2\) be first node encountered by DFS

- If \(x \in C_1\):
Run \( \text{DFS}(G) \), and let \( \text{finish}(C) = \max_{v \in C} \text{finish}(v) \)

**Lemma**

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

Let \( x \in C_1 \cup C_2 \) be first node encountered by DFS

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

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

**Lemma**

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

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

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

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

**Lemma**

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

Let $x \in C_1 \cup C_2$ be first node encountered by DFS
- If $x \in C_1$: all of $C_2$ reachable from $x$, so $\text{DFS}(x)$ does not complete until all of $C_2$ finished
- If $x \in C_2$: all of $C_2$ reachable from $x$ but nothing from $C_1$, so $x$ finishes before any node in $C_1$ starts
SCCs and DFS

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

Lemma

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

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

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

So node with max finish time in a source SCC. Want sink.
SCCs and DFS

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

Lemma

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

Let $x \in C_1 \cup C_2$ be first node encountered by DFS
- If $x \in C_1$: all of $C_2$ reachable from $x$, so DFS($x$) does not complete until all of $C_2$ finished
- If $x \in C_2$: all of $C_2$ reachable from $x$ but nothing from $C_1$, so $x$ finishes before any node in $C_1$ starts

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

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

Lemma

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

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

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

So node with max finish time in a source SCC. Want sink. Reverse all edges!
Kosaraju’s Algorithm

**Definition:** $G^T$ is $G$ with all edges reversed.

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

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

**Running Time:** $O(m + n)$
Kosaraju’s Algorithm

**Definition:** $G^T$ is $G$ with all edges reversed.

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

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

**Definition:** $G^T$ is $G$ with all edges reversed.

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

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

**Running Time:** $O(m + n)$
Correctness Sketch

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

**Theorem**

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

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

**Theorem**

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

Induction on $i$. 
Correctness Sketch

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

**Theorem**

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

Induction on $i$.

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

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

**Theorem**

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

Induction on \( i \).

**Base case:** \( i = 1 \). By previous argument, largest finishing time in \( G^T \Rightarrow \) in sink SCC of \( G \)
\[ \Rightarrow C_1 \text{ is sink SCC of } G \]

**Inductive case:** Let \( v \) node remaining with largest finishing time.

- By induction, current graph is \( G \) minus \( i - 1 \) SCCs of \( G \)
- Implies \( v \) 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