Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
A graph $G(V, E)$ is a pair of sets:

- $V$ is a set of nodes/vertices
- $E$ is a set of edges $(u,v)$ s.t $u,v \in V$
Examples

- **Social Networks:**
  - FB: $V$: people; $E$: $(u, v) \rightarrow u$ and $v$ are friends
  - Twitter: $V$: people/organizations; $E$: $(u, v) \rightarrow u$ follows $v$

- **World Wide Web:** $V$: web pages; $E$: $(u, v) \rightarrow$ page $u$ links to $v$

- **Molecular Networks:** $V$: atoms; $E$: $(u, v) \rightarrow$ bond btw $u$ and $v$

- **Many more …**
Some Graph Terminology (1)

- **Directed vs Undirected**
  - Directed: edges not (necessarily) symmetric
    - E.g. Twitter, WWW
  - Undirected: edges are symmetric \((u, v) \in E \Rightarrow (v, u) \in E\)
    - FB friendship graph, Road-maps \((V: \text{cities}, E: \text{roads})\)

- **Simple vs Multigraphs**
  - Simple: no parallel edges can exist between \((u, v)\)
  - Multigraphs: parallel edges are allowed
Some Graph Terminology (2)

- **Cyclic vs Acyclic**
  - Cyclic: can start from v; follow a path; and come back to v
  - Acyclic: no such cycles exist in the graph

- **Connected vs Unconnected**
  - Connected: graph in “one piece” (will elaborate more)
  - Unconnected: graph can be in “multiple pieces”
More Terminology & Conventions (1)

◆ Convention 1

◆ $|V| = n$

◆ $|E| = m$

◆ Note: $m$ is not a free parameter!

Q: In an undirected $G$ w/ $n$ vertices, what’s the max # edges?

A: $m \leq \binom{n}{2} = n(n-1)/2$

Q: In a directed $G$ w/ $n$ vertices, what’s the max # edges?

A: $m \leq 2n(n-1)/2 = n(n-1)$

Q: In an (undirected) connected $G$ w/ $n$ vertices, the min # edges?

A: $m \geq n-1$ (Exercise: Proof by induction on $n$)
More Terminology & Conventions (2)

◆ If \( m \) varies between \( O(n) \) and \( O(n^2) \)

\[
\log(m) = \Theta(\log(n))
\]

◆ Convention 2: We’ll always use \( \log(n) \) in analysis (not \( \log(m) \))

◆ E.g: Dijkstra’s Alg is \( O(n\log(n)) \) instead of \( O(n\log(m)) \).

◆ Degree of a vertex:

◆ Out Degree: \( \text{out-deg}(v) \): # outgoing edges of \( v \)
  ◆ i.e # \((v, w)\) \( \in E \)

◆ In Degree: \( \text{in-deg}(v) \): # incoming edges of \( v \)
  ◆ i.e # \((w, v)\) \( \in E \)

◆ Note: \( \text{deg}(v) \) usually means out-deg not in-deg
1. Adjacency Matrix: an n x n matrix A
   - $A[i, j ] = 1$ if $(i, j) \in E$
   - $A[i, j ] = 0$ otherwise
   - $O(n^2)$ storage.

   **Operations:**
   - Lookup $(u,v)$ exists: $O(1)$
   - Iterate over u’s out/in edges: $O(n)$
   - Usually good for very “dense” graphs, i.e.
     when most edges exist
2. Adjacency List

- $O(m + n)$ storage.
- Operations:
  - Lookup $(u,v)$ exists: $\text{deg}(u)$
  - Iterate over $u$’s out/in edges: $\text{deg}(u)$
- Usually good for “sparse” graphs

Mostly, we’ll be using Adj. List Format

If needed, can also store incoming edges in separate arrays
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
2 Basic Graph Traversal Algorithms

◆ BFS: Breadth-First Search Traversal
  ◆ Starts from s and traverses the graph in waves
  ◆ s->s’s first degree nbrs->s’s 2nd degree nbrs -> ...

◆ DFS: Depth-First Search Traversal
  ◆ From s tries to go “as far as” it can “as fast as” it can
  ◆ Backtracking when stuck
BFS

- **L0** (Not seen/visited)
- **A** (Currently traversing)
- **B** (Not seen/visited)
- **C** (Not seen/visited)
- **D** (Not seen/visited)
- **E** (Not seen/visited)
- **F** (Not seen/visited)
- **G** (Not seen/visited)
- **H** (Not seen/visited)
- **I** (Not seen/visited)

**Legend:**
- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
BFS

Not seen/visited

Currently traversing

Seen/Visited

Finished traversing
BFS
BFS

- **L0**: Not seen/visited
- **L1**: Seen/visited
- **L2**: Currently traversing
- **L3**: Finished traversing

Diagram:
- A (L0)
- B (L1)
- C (L1)
- D (L1)
- E (L0)
- F (L0)
- G (L0)
- H (L0)
- I (L0)
BFS

- **L0**: Not seen/visited
- **L1**: Currently traversing
- **L2**: Seen/Visited
- **L3**: Finished traversing

Nodes and their status:
- A: L0
- B: L1
- C: L1
- D: L1
- E: L2
- F: L2
- G: L2
- H: L2
- I: L2

Not seen/visited: A, B, E
Currently traversing: C
Seen/Visited: D, F, G
Finished traversing: H, I
BFS

Not seen/visited
Currently traversing
Seen/Visited
Finished traversing
BFS

- L0 (Not seen/visited)
- L1 (Currently traversing)
- L2 (Seen/Visited)
- L3 (Finished traversing)
BFS

- **L0**: Not seen/visited
- **L1**: Currently traversing
- **L2**: Seen/Visited
- **L3**: Finished traversing

Diagram:

- A (L0) connected to B (L1)
- B (L1) connected to C (L1)
- C (L1) connected to D (L1)
- D (L1) connected to E (L2)
- E (L2) connected to F (L2)
- G (L2) connected to H (L2)
- H (L2) connected to I (L3)
BFS

- **Not seen/visited**
- **Currently traversing**
- **Seen/Visited**
- **Finished traversing**
BFS

- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
BFS

Not seen/visited
Currently traversing
Seen/Visited
Finished traversing
BFS At a Higher Level

Not seen/visited

Currently traversing

Seen/Visited

Finished traversing
BFS At a Higher Level
BFS At a Higher Level
BFS At a Higher Level

- **Current Level (L1)**: A, B, D
- **Not seen/visited**: C, E, F, G, H, I
- **Currently traversing**: None
- **Seen/Visited**: None
- **Finished traversing**: None
BFS At a Higher Level

- **Not seen/visited**
- **Currently traversing**
- **Seen/Visited**
- **Finished traversing**
BFS At a Higher Level

- **L0** Not seen/visited
- **L1** Currently traversing
- **L2** Seen/Visited
- **L3** Finished traversing

Diagram:
- A (L0) connected to B (L1)
- D (L1) connected to C (L2) and E (L2)
- B (L1) connected to C (L2)
- G (L2) connected to H (L2)
- H (L2) connected to I (L3)
BFS At a Higher Level

- **Not seen/visited**: Blue
- **Currently traversing**: Green
- **Seen/Visited**: Yellow
- **Finished traversing**: Red

Graph showing BFS traversal levels:
- **L0**: A
- **L1**: B, D, C
- **L2**: C, G, H, I
- **L3**: I

Traversal order:
1. **A (L0)**
2. **B (L1)**
3. **C (L1)**
4. **D (L1)**
5. **E (L2)**
6. **F (L2)**
7. **G (L2)**
8. **H (L2)**
9. **I (L3)**
BFS At a Higher Level

Not seen/visited
Currently traversing
Seen/Visited
Finished traversing
**BFS Pseudocode**

1. procedure BFS(G(V, E), s)
2.   let Q be a new queue;
3.   mark s visited
4.   enqueue(s, Q)
5. while (Q not empty):
6.     let v = dequeue(Q)
7.     for each neighbor u of v:
8.       if (u is not-visited):
9.         mark u as visited;
10.        enqueue(u, Q)
11.       mark v as finished

**Total Runtime: O(n+m)**

(with adj list)

Total runtime: O(n+m) with adjacency list.

\[ \sum_{v=1}^{n} \text{out-deg}(v) = m \]

O(n)

1. procedure BFS(G(V, E))
2.   mark all vertices as not-visited
3.   for each (v ∈ V, if v is not-visited do BFS(G, v)

\[ O(n) \]
DFS
DFS

A — B — C — D — E

B

C — G — H — I

E — F

A is marked in green.
DFS
DFS

A

B

C

D

E

F

G

H

I
DFS

A

B

C

D

E

F

G

H

I
DFS
DFS
DFS
DFS
DFS
DFS
DFS
DFS
DFS
DFS

A

B

C

D

E

F

G

H

I
DFS

A

B

C

D

E

F

G

H

I
DFS
DFS Pseudocode (Recursive Version)

1. procedure DFS(G(V, E), s)
2. mark s visited
3. for each (neighbor v of s):
4.   if (v is not visited):
5.     DFS(G, v);
6. mark s finished

1. procedure DFS(G(V, E))
2. mark all vertices as not-visited
3. for each (v ∈ V, if v is not-visited do DFS(G, v)

Runtime: O(n + m) (with adj. list)

Visually:
- each v traversed once
- each (u, v) will be “traversed” at most twice:
  1. when attempting to visit v
  2. and possibly once when backtracking)
Properties of BFS/DFS

Key Observation 1:

*BFS/DFS are both linear time*

*when implemented by the right data structures*

Key Observation 2:

*A BFS/DFS starting from s will reach all vertices t such that s has a path to t.*

Exercise: Prove this claim by induction (on the length of the path that s has to t).
BFS Tree of a “Connected Graph”

Dfn “parent of v” $p(v)$:

Vertex $u$ that was being traversed \textit{when $v$ was first visited}

In simulation: the green vertex $u$ when $v$ became yellow

BFS Tree($V_T$, $E_T$) is the graph s.t.

- $V_T = V$

- $E_T = \{ \forall v: (p(v), v) \}$ (can define as $(v, p(v))$ as well, depending on a application or as undirected)
BFS Tree

Not seen/visited
Currently traversing
Seen/Visited
Finished traversing
BFS Tree

- L0: Not seen/visited
- L1: Currently traversing
- B: Seen/Visited
- C: Finished traversing

Diagram nodes:
- A
- B
- C
- D
- E
- F
- G
- H
- I
BFS Tree

- **L0** Not seen/visited
- **L1** Currently traversing
- **H** Seen/Visited
- **G** Finished traversing

Diagram:
- Node A connected to L0 and L1
- Node B connected to L0
- Node C connected to L1
- Node D connected to L1
- Node E connected to F
- Node H connected to G
- Node I connected to F

Steps:
1. Start at L0 (Not seen/visited)
2. Move to L1 (Currently traversing)
3. Visit H (Seen/Visited)
4. Finish traversing G (Finished traversing)
BFS Tree

- L0
- L1
- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing

Diagram of BFS Tree with nodes A, B, C, D, E, F, G, H, I.
BFS Tree

- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
BFS Tree

- **A**
  - **L0**
    - **B**
  - **L1**

- **D**
  - **L1**
    - **C**
      - **H**
        - **L2**
          - **I**
  - **E**
  - **F**

- **G**

**Colors**:
- Blue: Not seen/visited
- Green: Currently traversing
- Yellow: Seen/Visited
- Red: Finished traversing
BFS Tree

- **Not seen/visited**: Light blue
- **Currently traversing**: Green
- **Seen/Visited**: Yellow
- **Finished traversing**: Red

The graph shows a Breadth-First Search (BFS) traversal of a tree. The nodes are labeled with letters (A, B, C, D, E, F, G, H, I) and are colored according to their traversal status.
BFS Tree

- **A**: L0
- **B**: L1
- **C**: L1
- **D**: L1
- **E**: L2
- **F**: L2
- **G**: L2
- **H**: L2
- **I**: L2

**Colors and States**:
- **Red**: Not seen/visited
- **Blue**: Currently traversing
- **Green**: Seen/Visited
- **Yellow**: Finished traversing
BFS Tree

Not seen/visited
Currently traversing
Seen/Visited
Finished traversing
BFS Tree

- **L0**: Not seen/visited
- **L1**: Currently traversing
- **L2**: Seen/Visited
- **L3**: Finished traversing

Nodes:
- A
- B
- C
- D
- E
- F
- G
- H
- I
BFS Tree

A
L0

B
L1

D
L1

E
L2

F
L3

G
L2

H
L2

C
L1

I
L3

- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
BFS Tree

- **Not seen/visited**: Blue
- **Currently traversing**: Green
- **Seen/Visited**: Yellow
- **Finished traversing**: Red

Diagram shows a BFS tree with nodes L0, L1, L2, L3, A, B, C, D, E, F, G, H, I.
BFS Tree

- **L0**: Not seen/visited
- **L1**: Not seen/visited
- **L2**: Currently traversing
- **L3**: Seen/Visited
- **L4**: Finished traversing

Nodes and edges represent the traversal process in a breadth-first search (BFS) tree. The coloring indicates the state of each node during the traversal.
BFS Tree

- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
BFS Tree

- L0
- L1
- L2
- L3

- A
- B
- C
- D
- E
- F
- G
- H
- I

- Not seen/visited
- Currently traversing
- Seen/Visited
- Finished traversing
***Note***

Let level of $s$ be 0

Level of a $v$ is $= 1 + \text{level}(p(v))$
BFS Tree

Suppose this is the BFS-Tree of a (directed) G.
Suppose this is the BFS-Tree of a (directed) $G$.

Q: Can “forward” edge, i.e. btw non-consecutive levels, exist in $G$?

A: No

B/c 9 and 17 would be levels 1 and 2, respectively
Suppose this is the BFS-Tree of a (directed) G.

Q: Can “forward” edge, i.e. btw non-consecutive levels, exist in G?
A: No

B/c 9 and 17 would be levels 1 and 2, respectively.

Q: Can “cross” edge, i.e. btw same levels, exist in G?
A: Yes
Suppose this is the BFS-Tree of a Graph G.

Q: Can “forward” edge, i.e. btw non-consecutive levels, exist in G?
   A: No
   B/c 9 and 17 would be levels 1 and 2, respectively

Q: Can “cross” edge, i.e. btw same levels, exist in G?
   A: Yes

Q: Can “back” edge, i.e. from larger to smaller level, exist in G?
   A: Yes in a directed graph. No in an undirected graph.
DFS Tree of a “Connected Graph”

◆ Can be defined in the exact same way as BFS-tree

◆ Exercise: Do the analysis of what types of edges can exist given a DFS-Tree of a graph G.
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
Unweighted Single Source Shortest Paths

- **Input:** $G(V, E)$ (directed or undirected), source vertex $s$
- **Output:** $\forall v \in V$, shortest path and shortest $\text{dist}(s, v)$
Shortest Path Example
Shortest Path Example

Q: What’s dist(S, A)?

A: 1 (Path: S->A)
Q: What’s dist(S, Z)?

A: 2 (Path: S→Y→Z)
Q: What’s dist(S, E)?

A: 2 (Path: S->B->Z)
Shortest Path Example

Q: What’s dist(S, D)?

A: 3

Paths: S->B->E->D or S->A->C->D
Just Run BFS!

The level numbers will exactly be the distances!
Shortest Path Example
Shortest Path Example
Shortest Path Example
Shortest Path Example

The diagram illustrates a network of nodes (Z, X, Y, S, A, B, C, D, E) with weighted edges connecting them. The weights are indicated on the edges. The shortest path from Z to E is highlighted in red.
Shortest Path Example

Graph with nodes labeled Z, X, Y, S, A, B, C, D, E and edges marked with weights 0, 1, 2, and 2.
Shortest Path Example
Shortest Path Example
Formal correctness proof is by induction on the length of the \((s, t)\) paths. (Check)

Runtime: \(O(n + m)\)

In linear time, all shortest paths from \(s\) to every other vertex in \(V\)!
The BFS tree can be used to construct actual paths

Just follow your parent! (suppose we stored \((v, p(v))\) direction)
The BFS tree can be used to construct actual paths.

Just follow your parent! (suppose we stored \((v, p(v))\) direction)

Ex: What’s the path from S to D?
Just follow back from D on BFS Tree.
Using The BFS Tree To Construct Paths

- The BFS tree can be used to construct actual paths
- Just follow your parent! (suppose we stored \((v,p(v))\) direction)

Ex: What’s the path from S to D?
Just follow back from D on BFS Tree.
The BFS tree can be used to construct actual paths
Just follow your parent! (suppose we stored \((v,p(v))\) direction)

Ex: What’s the path from S to D?
Just follow back from D on BFS Tree.

C -> D
The BFS tree can be used to construct actual paths

Just follow your parent! (suppose we stored \((v, p(v))\) direction)

Ex: What’s the path from S to D?

Just follow back from D on BFS Tree.

A→C→D
The BFS tree can be used to construct actual paths

Just follow your parent! (suppose we stored \((v, p(v))\) direction)

Ex: What’s the path from S to D?
Just follow back from D on BFS Tree.

\[ S \rightarrow A \rightarrow C \rightarrow D \]
The BFS tree can be used to construct actual paths

Just follow your parent! (suppose we stored \((v,p(v))\) direction)

Constructs a path \((s, t)\) in \(d(s,t)\) time!
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. **Application 2: Bipartiteness/2-coloring**
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
Bipartite Checking/2-coloring

**Input:** undirected $G(V, E)$

**Output:** True/False to the following question:

Can $V$ be partitioned as $V_1$ (red) and $V_2$ (green)

s.t. $\forall (u,v) \in E$, $u$ is red and $v$ is green
Is this graph 2-colorable?
Bipartite Checking/2-coloring

Yes
Is this graph 2-colorable?
Bipartite Checking/2-coloring

Yes
Is this graph 2-colorable?
Bipartite Checking/2-coloring

Is this graph 2-colorable?
Is this graph 2-colorable?
Bipartite Checking/2-coloring

Is this graph 2-colorable?
Bipartite Checking/2-coloring

Not colorable.
Observation

Suppose we do BFS from s and assign s (w.l.o.g) the color red.

Then if G is 2-colorable:

s is level 0 and colored red.

Then all of level 1 vertices has to be green.

Then all of level 2 has to be red.

Then all of level 3 has to be green.

Then all of level 4 has to be red.

...

So let’s “tentatively” color vertices according to the levels of the vertices in the BFS-tree.
What if G is not colorable?

Then there must be a “cross edge” in the graph.
So let’s just check if a cross edge breaks our coloring.
BFS-based Bipartite Checking Algorithm

1. procedure Bipartite-Checking(undir. G(V, E))
2. run BFS(G(V,E))
3. give even levels red; odd levels green
4. if ∃(u, v) ∈ E s.t. u & v are the same color
   return false
5. else
6. return true

Runtime: O(n + m)
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
Undirected (Weakly) Connected Components

◆ Input: undirected $G(V, E)$
◆ Output: $CC_1, CC_2, ..., CC_k$

where each $CC_i$ is a maximal subset of $V$, s.t.

- each $s, t$ in $CC_i$ has a path to each other AND
- each $v \in V$ is part of one $CC_i$. 
Undirected Connected Components

CC_1

CC_2

CC_3

CC_4
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation

A

B

D

H

C

L

G

J

M

D

I

K

F
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
**Run BFS Just With Label Propagation**

1. procedure BFS-LP(G(V, E), s)
2. let Q be a new queue;
3. mark s visited
4. label s with s
5. enqueue(s, Q)
6. while (Q not empty):
7. let v = dequeue(Q)
8. for each neighbor u of v:
9. if (u is not-visited):
10. mark u as visited;
11. label u with s
12. enqueue(u, Q)
13. mark v as finished

**Runtime: O(n+m)**
(with adj list)

---

1. procedure BFS-LP(G(V, E))
2. mark all vertices as not-visited
3. for each v ∈ V, if v is not-visited do BFS-LP(G, v)
Note that using BFS was not critical.

We could have easily propagated the labels using DFS.
Outline For Today

1. Graph Terminology
2. BFS/DFS & BFS/DFS Tree
3. Application 1: Unweighted Single Source Shortest Paths
4. Application 2: Bipartiteness/2-coloring
5. Application 3: Connected Comp. in Undir. Graphs
6. Application 4: Topological Sort
**Directed Acyclic Graphs (DAGs)**

- Graphs to model “dependency”/“prerequisite” relationships
  - directed & acyclic
Example #1

- Job Scheduling in Operating Systems

```
Example #1

- grep
- job1
- tee
- wc
- job2
- cat
- awk
- echo
```
Example #2

◆ CS Course Dependencies

Why Acyclic?
DAG Terminology

- **source**: no incoming edges
- **sink**: no outgoing edges
DAG Terminology

- **source**: no incoming edges
- **sink**: no outgoing edges
DAG Properties

◆ Property 1: Every DAG has at least 1 source

◆ Proof:

Assume contrary: then every \( v \) has at least 1 incoming edge
Start at \( v_1 \) and follow, repeatedly, one of \( v_1 \)'s in-edges

Cycle

◆ Property 2: Every DAG has at least 1 sink: (similar proof)
Topological Sorting of a DAG

- **Input:** A $G(V, E)$ which is a DAG
- **Output:** vertices in sorted order s.t
  - each vertex $v$ appears after its dependencies
- **A More Formal Definition:**

  Given an input DAG $G(V, E)$ order the nodes such that
  if $(u, v) \in E$, then $u$ appears before $v$
Example

Input:

Output:

Output is not unique
Topological Sorting Algorithm #1

```plaintext
procedure topologicalSort1(DAG G):
    let result empty list
    while G is not empty:
        1. let v be a source node in G
        2. add v to result
        3. remove v from G
    return result/reverse

Note: You can also pick a sink iteratively!
```
Algorithm #1 Simulation

Input:
- CS 106A
- CS 106B
- CS 103

Output:
- CS 107
- CS 109
- CS 110
- CS 143
- CS 161
Algorithm #1 Simulation

Input:

Output:
Algorithm #1 Simulation

Input:

Output:

106A
Algorithm #1 Simulation

Input:

CS 106B → CS 107 → CS 110
CS 103 → CS 109 → CS 161

Output:

106A
Algorithm #1 Simulation

Input:

- CS 106B
- CS 107
- CS 110
- CS 143
- CS 109
- CS 161

Output:

- 106A
Algorithm #1 Simulation

Input:

CS 106B → CS 107 → CS 110

CS 103 → CS 109 → CS 161

Output:

106A  106B
Algorithm #1 Simulation

Input:

CS 103

CS 107 → CS 110
CS 143
CS 109 → CS 161

Output:

106A 106B
Algorithm #1 Simulation

Input:

Output:

106A  106B
Algorithm #1 Simulation

Input:

Output:

106A  106B  103
Algorithm #1 Simulation

**Input:**
- CS 107
- CS 110
- CS 143
- CS 161

**Output:**
- 106A
- 106B
- 103
Algorithm #1 Simulation

Input:

Output:

106A  106B  103
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107
Algorithm #1 Simulation

Input:
- CS 110
- CS 143
- CS 161

Output:
- 106A
- 106B
- 103
- 109
- 107
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107

CS 110  CS 143  CS 161
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161

CS 110
CS 143
CS 161
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161

CS 110

CS 143
Algorithm #1 Simulation

Input:

Output:

CS 110

CS 143

106A 106B 103 109 107 161
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161  143

CS 110

CS 143
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161  143

CS 110
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161  143  CS 110
Algorithm #1 Simulation

Input:

Output:

106A   106B   103   109   107   161   143   110

CS 110
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161  143  110
Algorithm #1 Simulation

Input:

CS 106A → CS 106B → CS 107 → CS 110
CS 106A → CS 103 → CS 109 → CS 161
CS 106A → CS 103
CS 106A → CS 109
CS 106A → CS 110
CS 106A → CS 143
CS 106A → CS 161

Output:

106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
106A → 106B → 103 → 109 → 107 → 161 → 143 → 110
Runtime of TS Algorithm #1

- Depends on implementation

```plaintext
procedure topologicalSort1(DAG G):
    let result empty list.
    while G is not empty:
        1. let v be a source node in G
        2. add v to result
        3. remove v from G
    return result
```

Naïve Implementation: find source by looping over V

Runtime: \(O(n^2)\)

**Exercise:** *Can do it in: \(O(n + m)\).*
TS Algorithm #2 (Using DFS)

- Run DFS
- Keep track of the “finishing times” of the vertices

Finishing Time $f[v]$: the time when $v$ turns red
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
TS Algorithm #2 (Using DFS) Simulation 1
Order By Decreasing Finishing Times (1)
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2

Diagram:
- Nodes: H, A, C, B, E, G, D, F
- Edges:
  - H -> 6
  - 6 -> A
  - A -> 5
  - 5 -> C
  - C -> 3
  - E (green) -> G
  - G -> 7
  - 7 -> D
  - D -> 2
  - 2 -> F
  - F -> 1
TS Algorithm #2 (Using DFS) Simulation 2

Diagram representation of the TS Algorithm using Depth First Search (DFS) simulation.
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2
TS Algorithm #2 (Using DFS) Simulation 2

Diagram of a directed graph with nodes labeled from 1 to 8, and arrows indicating the direction of the edges.
Order By Decreasing Finishing Times (2)
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3

Diagram of a graph with nodes labeled E, H, A, C, B, 1, G, D, F, and G. The diagram illustrates a network with directed edges connecting these nodes.
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3

Diagram: A graph with nodes labeled E, H, G, D, F, B, C, and numbers 1, 2, 3, 4. Nodes are connected by directed edges.
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
TS Algorithm #2 (Using DFS) Simulation 3
Order By Decreasing Finishing Times (3)
3 DFS Simulations 3 Topological Orders
procedure topologicalSort2(DAG G):
    run DFS(G) & put each finished vertex
    into an array in reverse order

Runtime = Runtime of DFS = $O(n + m)$

Correctness?
**DFS PseudoCode With Finishing Time Keeping**

```
global var t = 1;
procedure DFS(G):
    f: array of size n initialized to null
    for i = 1 to n:
        if V[i] is not yet visited:
            DFS(G, i)

procedure DFS(G, u):
    mark u as visited
    for all neighbors v of u:
        if v is not yet visited:
            DFS(G, v)
    f[u] = t; t++;
```
**Key Claim About Finishing Times in DAGs**

*In a DAG if $u \rightarrow v$, then $f[u] > f[v]*

**Proof:**

Break into 2 cases by whether $u$ or $v$ is visited first

**Case 1:** $u$ is visited first. Then:
- DFS($v$) call will be made before DFS($u$) finishes /
  *(b/c DFS call is made on every edge from $u$)*
- $\Rightarrow f[u] > f[v]$

**Case 2:** $v$ is visited first. Then:
- Recall the graph is a acyclic
- Therefore DFS cannot discover $u$ from $v$.
- Therefore $v$ has to finish before even discovering $u$
- $\Rightarrow f[v] < s[u] < f[u]$ *(where $s[u]$ is discovery time of $u$)*
- $\Rightarrow f[u] > f[v]$

*In a DAG if $u \rightarrow v$, then $f[u] > f[v]*
Extended Key Claim

*In a DAG if \( u \sim v \), then \( f[u] > f[v] \)*

Proof: By induction on the length of the paths, where
Base case is simply Key Claim from previous slide
Prove it as an Exercise!

**Note: Not needed for the correctness of Topological Sort with DFS finishing times but will be important next lecture.**
Correctness Proof: Immediate from Key Claim

if \((u, v)\) exists, then \(f(u) > f(v)\)

we order according to decreasing \(f\) values

therefore \(u\) will be ordered before \(v\)