Lecture 13-14: BFS/DFS Applications

CS 341: Algorithms
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 \((u, v) \in E \implies (v, u) \in E\)
  - 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: $m \leq \binom{n}{2} = \frac{n(n-1)}{2}$

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 $n$)
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: $\deg(v)$ usually means out-deg not in-deg
1. Adjacency Matrix: an $n \times n$ 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 2\textsuperscript{nd} 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

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

Diagram showing a Breadth-First Search (BFS) traversal of a graph with nodes labeled A to I.
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

Graph representation:

- **L0**: Not seen/visited
- **L1**: Currently traversing
- **Not seen/visited**: Currently not in the traversal sequence
- **Seen/Visited**: Nodes already visited or currently being traversed
- **Finished traversing**: Nodes that have been fully explored

Nodes:

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

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

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

Nodes:
- A
- B
- C
- D
- E
- F
- G
- H

Connections:
- A to B
- B to C
- C to L1
- L1 to D
- D to E
- E to F
- F to G
- G to H
- H to L2
- L2 to I
BFS

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

The diagram illustrates the Breadth-First Search (BFS) traversal of a graph, with nodes colored to indicate their status during the traversal process.
BFS

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

A
L0
B
L1
C
D
L1
E
L2
F

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

A -> L0 -> B
D -> L1 -> C
E -> L2

BFS algorithm:
- 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

- **LO**: Not seen/visited
- **Currently traversing**: Green
- **Seen/Visited**: Yellow
- **Finished traversing**: Red
BFS At a Higher Level

- **A**: L0 (Not seen/visited)
- **B**: L1 (Not seen/visited)
- **C**: L1 (Currently traversing)
- **D**: L1 (Not seen/visited)
- **E**: (Not seen/visited)
- **F**: (Not seen/visited)
- **G**: (Not seen/visited)
- **H**: (Not seen/visited)
- **I**: (Not seen/visited)

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

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

- **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

- **L0**
- **L1**
- **L2**
- **L3**

**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)

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

$O(n)$

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

A — B — C — D — E

B — C — G — H — I
DFS
DFS
DFS

A - B - C - D - E
B - D - C - H - I
DFS
DFS
DFS
DFS
DFS
DFS

A -- B -- C -- D -- E

F -- G -- H -- I
DFS
DFS
DFS
DFS

A - B - C - D - E

B - 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 seen/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

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

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

Nodes and edges illustrate the breadth-first search traversal process.
BFS Tree

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

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

- **L0** (Not seen/visited) - A
- **L1** (Not seen/visited) - B, C
- **L1** (Seen/Visited) - D, E, F
- **L1** (Currently traversing) - C
- **L1** (Finished traversing) - C

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

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

Diagram:
- A (L0) to B (L1) (Red, Finished traversing)
- B (L1) to C (L1) (Red, Finished traversing)
- C (L1) to D (L1) (Blue, Currently traversing)
- C (L1) to E (Blue)
- C (L1) to F (Blue)
- C (L1) to G (Blue)
- C (L1) to H (Blue)
- C (L1) to I (Blue)
BFS Tree

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

Nodes: A, B, C, D, E, F, G, H, I

Connections:
- A to B
- B to C
- C to H, D
- D to E, I, G
- E to F
- F to G, I
- G to H, I
- H to I

Levels:
- L0: A
- L1: B, C, D, E, F
- L2: G, H, I
BFS Tree

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

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

Diagram:
- A (L0, seen/visited)
- B (L1, not seen/visited)
- C (L1, currently traversing)
- D (L1, seen/visited)
- E (seen/visited)
- F (not seen/visited)
- G (L2, currently traversing)
- H (L2, seen/visited)
- I (not seen/visited)
BFS Tree

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

Diagram mostrating BFS tree with nodes A, B, C, D, E, F, G, H, I, and their traversal status.
BFS Tree

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

The image represents a breadth-first search (BFS) tree with the following nodes and colors:

- **A** (L0): Not seen/visited
- **B** (L1): Not seen/visited
- **C** (L1): Currently traversing
- **D** (L1): Seen/Visited
- **E** (L2): Finished traversing
- **F** (L2): Not seen/visited
- **G** (L2): Seen/Visited
- **H** (L2): Seen/Visited
- **I** (L2): Finished traversing
BFS Tree

- A
- L0
- L1
- B
- C
- L1
- D
- L1
- E
- L2
- F
- G
- L2
- H
- L2
- L3
- I

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

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

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

A - L0
  B - L1
  C - L1
  D - L1
  E - L2
  F - L2

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

- **Not seen/visited**
- **Currently traversing**
- **Seen/Visited**
- **Finished traversing**
***Note***
Let level of $s$ be $0$

Level of a $v$ is $= 1 + \text{level}(p(v))$
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/cause 9 and 17 would be levels 1 and 2, respectively
BFS Tree

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. In an undirected graph (9, 5) edge cannot but the other one can.
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
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)
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

Graph showing the shortest path from 'S' to 'E' with the label '0' marking the shortest path length.
Shortest Path Example
Shortest Path Example

![Graph with vertices Z, X, Y, S, B, A, C, D, E and edges with weights 1, 0, 1, 1, 1]
Shortest Path Example

Graph representation with nodes and edges labeled with weights.
Shortest Path Example
Shortest Path Example

Diagram of a network with nodes labeled Z, X, Y, S, A, B, C, D, and E, connected by edges with weights.

- Z to X: 2
- Z to Y: 2
- X to S: 2
- S to A: 1
- A to C: 2
- C to D: 3
- D to E: 2

The path from Z to E is Z -> Y -> S -> A -> C -> D -> E with a total weight of 11.
Shortest Path Example

![Graph Diagram]

- Vertices: Z, X, Y, S, A, B, C, D, E
- Edges with weights:
  - Z to X: 2
  - X to S: 2
  - S to Y: 1
  - Y to A: 1
  - A to B: 1
  - B to C: 2
  - C to D: 3
  - D to E: 2
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.)
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.
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.

\(A\rightarrow C\rightarrow D\)
The BFS tree can be used to construct actual paths

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

Ex: What’s the path from S to D?

Just follow back from D on BFS Tree.

\[ S \rightarrow A \rightarrow C \rightarrow D \]
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)

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

Not colorable.
Observation

Suppose we do BFS from $s$ and assign $s$ \textit{(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
5.    return false
6. else
7.    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:** $CC_1, CC_2, \ldots, CC_k$

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

Diagram: A B C D E F G H I J K L M
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation
BFS - Con. Comp. w/ Label Propagation

Diagram showing a graph with nodes labeled A, B, C, D, E, F, G, H, I, J, K, and L connected by edges, with different colors indicating different sets or labels.
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
Run BFS Just With Label Propagation

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

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

1. procedure BFS-LP\( (G(V, E)) \)
2. \[\text{mark all vertices as not-visited}\]
3. \[\text{for each } v \in V, \text{ if } v \text{ is not-visited } \text{ do BFS-LP}(G, v)\]
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

- Job Scheduling in Operating Systems

```

```
Example #1

- Job Scheduling in Operating Systems

```
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 Diagram]

- grep
- job1
- tee
- wc
- job2
- cat
- awk
- echo
**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

**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:

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

Output:

Output is not unique
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 107 -> CS 110
CS 106A -> CS 103 -> CS 109 -> CS 161
CS 103 -> CS 143
CS 109 -> CS 143

Output:
Algorithm #1 Simulation

Input:

CS 106A → CS 106B → CS 107 → CS 110

CS 103 → CS 109 → CS 161

Output:
Algorithm #1 Simulation

Input:
- CS 106B
- CS 103

Output:
- CS 107
- CS 109
- CS 143
- CS 110
- CS 161

106A
Algorithm #1 Simulation

Input:

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

Output:

106A
Algorithm #1 Simulation

Input:

Output:

106A  106B
Algorithm #1 Simulation

Input:

Output:

106A  106B
Algorithm #1 Simulation

Input:

Output:

106A  106B
Algorithm #1 Simulation

Input:

Output:

106A  106B  103
Algorithm #1 Simulation

Input:

Output:

106A  106B  103
Algorithm #1 Simulation

Input:

Output:

106A  106B  103
Algorithm #1 Simulation

Input:

Output:

- CS 107
- CS 109
- CS 110
- CS 143
- CS 161

106A  106B  103  109
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109
Algorithm #1 Simulation

**Input:**

- CS 107
- CS 110
- CS 143
- CS 161

**Output:**

- 106A
- 106B
- 103
- 109
Algorithm #1 Simulation

Input:

Output:

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

Input:

Output:
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  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:
Algorithm #1 Simulation

Input:

Output:

106A  106B  103  109  107  161  143

CS 110

CS 143
Algorithm #1 Simulation

Input:

Output:

CS 110

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

**Input:**

- 106A
- 106B
- 103
- 109
- 107
- 161
- 143
- 110

**Output:**

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

Input:

Output:
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: $O(n^2)$

Exercie: 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

Diagram of a network with nodes A, B, C, D, E, F, G, and H, connected by directed edges.
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

Diagram of a network with nodes labeled A, B, C, D, E, G, H, and F, and edges connecting them. The nodes A, B, and C are colored red, while the rest are blue. The node D is colored green, indicating a special status or role in the simulation.
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
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, E, G, D, B, F
- Edges: H to 6, 6 to 5, A to 5, 5 to 3, C to 3, E to 8, 8 to G, G to 7, 7 to 2, D to 2, F to 1, B to 4
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
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
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

Diagram showing three different topological orders of nodes E, H, A, C, G, D, F, B, with numbers 8, 7, 6, 5, 4, 3, 2, 1 assigned to each node.
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?
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 $u$)
$\Rightarrow f[u] > f[v]$
Extended Key Claim

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

Proof: Same argument as before and the observation that if $u$ is visited first, then any other $v$ that $u$ depends on will be visited before $u$ finishes (can be proved formally by induction on the length of $u \sim v$ paths)

Prove it as an Exercise!

**Note: Not needed for the correctness of Topological Sort with DFS finishing times but will be important for the next algorithm.**
Correctness of TS Algorithm #2

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 \(v\)