

Topological Sort and Graph Traversals

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao

Howard Xiao

Lucy Jiang

Chris Choi

Jade Watkins

Nathan Lipiarski

Ethan Knutson

Khushi Chaudhari

Richard Jiang

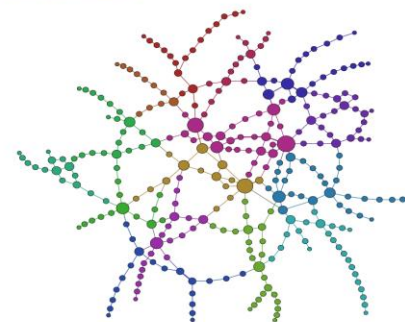
Poll Everywhere

pollev.com/cse332

Warm-up: this schematic map of the Paris Métro (Paris subway) is a graph. It exhibits the following characteristics:

- A. Undirected / Connected / Cyclic / Vertex-labeled
- B. Directed / Connected / Cyclic / Vertex-labeled
- C. Undirected / Connected / Cyclic / Edge-labeled
- D. Directed / Connected / Cyclic / Edge-labeled
- E. I'm not sure ...

Introduction to **Network Visualization** with GEPHI – Martin Grandjean
Examples



Announcements

- ❖ Schedule for the next few days:
 - Saturday: regular office hours
 - Monday: No lecture and no office hours (Memorial Day)
 - Tuesday: P3 checkpoint
 - This is a file upload, not a survey!
 - TAs will give you feedback on your ParallelSearcher
 - Tag your partner!
 - Wednesday: Exs 10-13 *actually* due

Lecture questions: pollev.com/cse332

Learning Objectives

- ❖ Be able to compare and choose graph representations
- ❖ Implement a topological sort
- ❖ Characterize the difference between depth-first and breadth-first searches in graphs
 - Also understand the tree-specific variants

Lecture Outline

- ❖ Graph Representations, cont.
 - **Review: Adjacency Matrix**
 - Adjacency List

- ❖ Topological Sort

- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - Conclusion

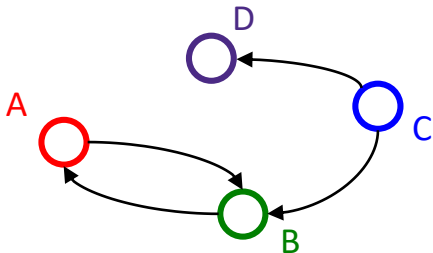
Lecture questions: pollev.com/cse332

What is the Data Structure?

- ❖ Is a Graph an ADT? Maybe!
 - “Develop an algorithm over the graph, then use whatever data structure is efficient”
- ❖ The “best” data structure can depend on:
 - Properties of the graph (e.g., dense versus sparse)
 - Common queries
 - e.g., “is (\mathbf{u}, \mathbf{v}) an edge?” vs “what are the neighbors of node \mathbf{u} ?”
- ❖ There are two standard graph representations:
 - *Adjacency Matrix* and *Adjacency List*
 - Different trade-offs, particularly time vs space

Adjacency Matrix: Representation

- ❖ Assign each node a number from 0 to $|\mathcal{V}| - 1$
- ❖ Graph is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix (ie, 2-D array) of booleans
 - $M[u][v] == \text{true}$ means there is an edge from u to v

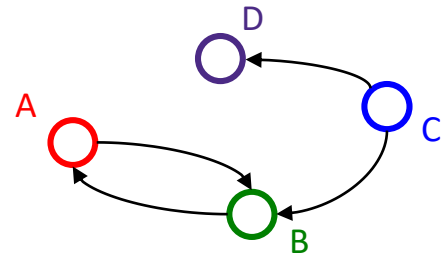


	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix: Properties (1 of 3)

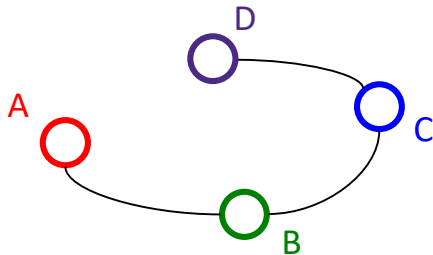
- ❖ Running time to:
 - Get a vertex's out-edges:
 - $O(|V|)$
 - Get a vertex's in-edges:
 - $O(|V|)$
 - Decide if some edge exists:
 - $O(1)$
 - Insert an edge:
 - $O(1)$
 - Delete an edge:
 - $O(1)$
- ❖ Space requirements:
 - $|V|^2$ bits
- ❖ Best for sparse or dense graphs?
 - Best for dense graphs

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F



Adjacency Matrix: Properties (2 of 3)

- ❖ How does the adjacency matrix vary for an *undirected graph*?
 - *Undirected graphs are symmetric about diagonal axis*
 - *Languages with array-of-array matrix representations can save ½ the space by omitting the symmetric half*
 - *Languages with “proper” 2D matrix representations (eg, C/C++) can’t do this*

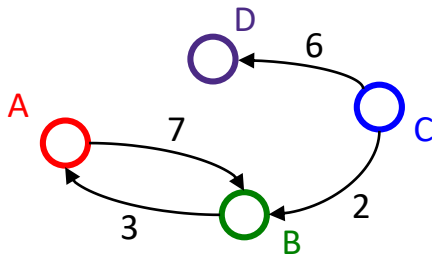


	A	B	C	D
A	F	T	F	F
B	T	F	T	F
C	F	T	F	T
D	F	F	T	F

this data is redundant / unnecessary

Adjacency Matrix: Properties (3 of 3)

- ❖ How can we adapt the representation for *weighted graphs*?
 - *Store the weight in each cell*
 - *Need some value to represent “not an edge”*
 - *In some situations, 0 or -1 works*



	A	B	C	D
A	0	7	0	0
B	3	0	0	0
C	0	2	0	6
D	0	0	0	0

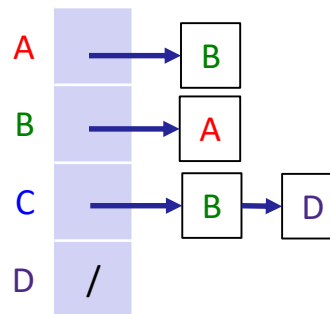
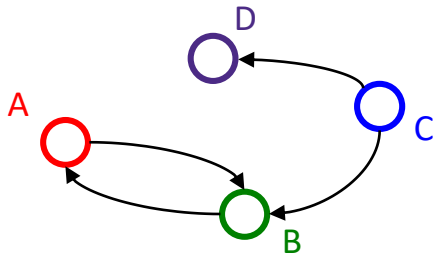
Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - **Adjacency List**
- ❖ Topological Sort
- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - Conclusion

Lecture questions: pollev.com/cse332

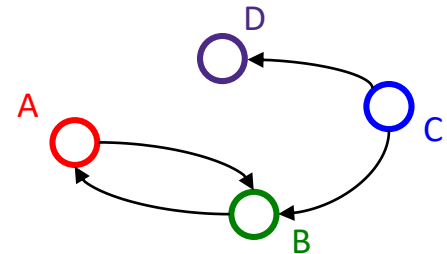
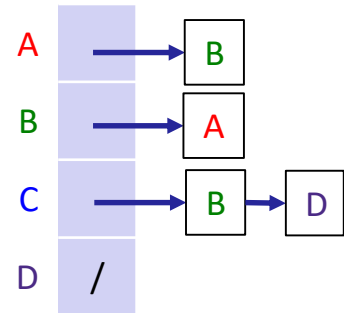
Adjacency List: Representation

- ❖ Assign each node a number from 0 to $|\mathcal{V}| - 1$
- ❖ Graph is an array of length $|\mathcal{V}|$; each entry stores a list of all adjacent vertices
 - E.g. linked list



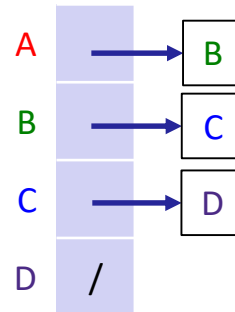
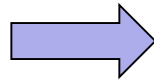
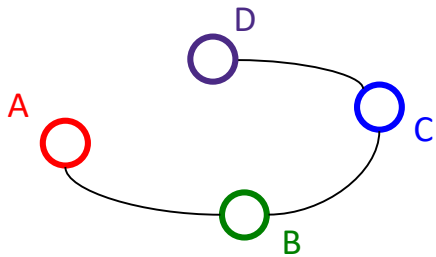
Adjacency List: Properties (1 of 3)

- ❖ Running time to:
 - Get a vertex's out-edges:
 - $O(d)$ where d is out-degree of vertex
 - Get a vertex's in-edges:
 - $O(|V| + |E|)$
 - (but could keep a second adjacency list for this!)
 - Decide if some edge exists:
 - $O(d)$ where d is out-degree of source vertex
 - Insert an edge:
 - $O(1)$
 - (unless you need to check if it's there; then $O(d)$)
 - Delete an edge:
 - $O(d)$ where d is out-degree of source vertex
- ❖ Space requirements:
 - $O(|V| + |E|)$
- ❖ Best for sparse or dense graphs?
 - Best for sparse graphs, so usually just stick with linked lists for the buckets

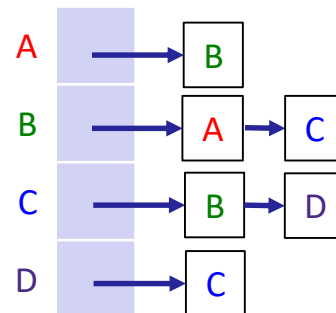


Adjacency List: Properties (2 of 3)

- ❖ How does the adjacency list vary for an *undirected graph*?
 - *Optionally, can double the entries to increase edge lookup speed*

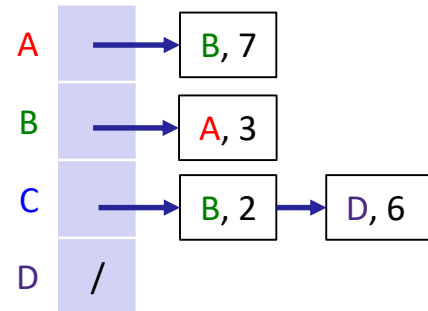
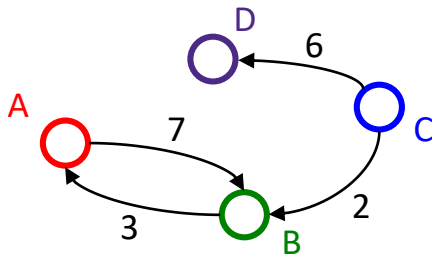


... or ...



Adjacency List: Properties (3 of 3)

- ❖ How can we adapt the representation for *weighted graphs*?
 - *Store the weight alongside the destination vertex*
 - *No need for a special value to represent “not an edge”!*



Summary: Which is Better?

- ❖ Graphs are often sparse:
 - Road networks are often grids
 - Every corner isn't connected to every other corner
 - Airlines rarely fly to all possible cities
 - Or if they do it is to/from a hub
- ❖ Adjacency lists should generally be your default choice
 - Slower performance compensated by greater space savings
 - Many graph algorithms rely heavily on `getAllEdgesFrom(v)`

	<code>getAllEdgesFrom(v)</code>	<code>hasEdge(v, w)</code>	<code>getAllEdges()</code>
Adjacency Matrix	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
Adjacency List	$\Theta(\text{degree}(v))$	$\Theta(\text{degree}(v))$	$\Theta(E + V)$

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List
- ❖ **Topological Sort**
- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - Conclusion

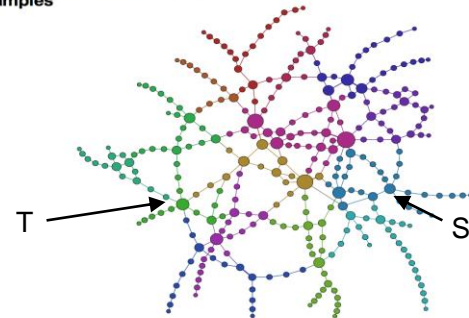
Lecture questions: pollev.com/cse332

Graph Queries

- ❖ Lots of interesting questions we can ask about a graph:
 - What is the shortest route from S to T? What is the longest route without cycles?
 - Are there cycles in this graph?
 - Is there a cycle that uses each *vertex* exactly once?
 - Is there a cycle that uses each *edge* exactly once?

Introduction to **Network Visualization** with GEPHI – Martin Grandjean

Examples



Graph Queries More Theoretically

- ❖ Some well known graph problems and their common names:
 - **s-t Path.** Is there a path between vertices s and t ?
 - **Connectivity.** Is the graph connected?
 - **Biconnectivity.** Is there a vertex whose removal disconnects the graph?
 - **Shortest s-t Path.** What is the shortest path between vertices s and t ?
 - **Cycle Detection.** Does the graph contain any cycles?
 - **Euler Tour.** Is there a cycle that uses every edge exactly once?
 - **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?
 - **Planarity.** Can you draw the graph on paper with no crossing edges?
 - **Isomorphism.** Are two graphs the same graph (in disguise)?
- ❖ Often can't tell how difficult a graph problem is without very deep consideration.

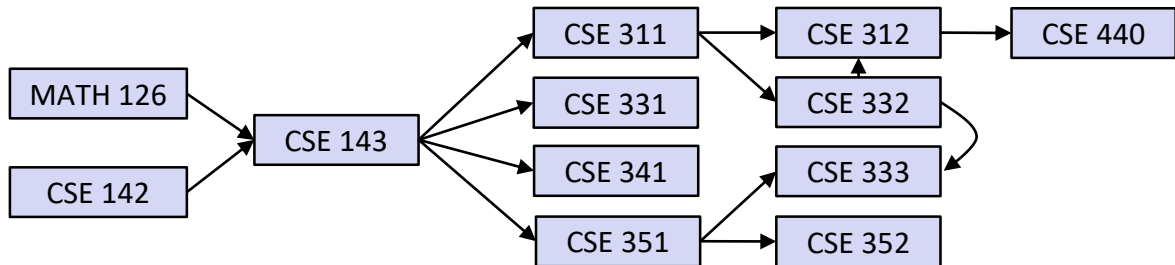
Graph Problem Difficulty

- ❖ Some well known graph problems:
 - **Euler Tour:** Is there a cycle that uses every *edge* exactly once?
 - **Hamilton Tour:** Is there a cycle that uses every *vertex* exactly once?
- ❖ Difficulty can be deceiving
 - An efficient Euler tour algorithm $O(\# \text{ edges})$ was found as early as 1873 [\[Link\]](#).
 - Despite decades of intense study, no efficient algorithm for a Hamilton tour exists. Best algorithms are exponential time.
- ❖ Graph problems are among the most mathematically rich areas of CS theory

Disclaimer: Do not use for official advising purposes!
Falsely implies CSE 332 is a prereq for CSE 312, etc.

Topological Sort

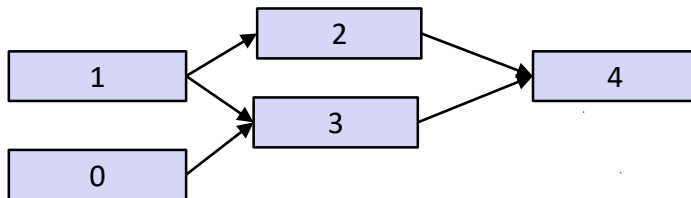
- ❖ Given a DAG, output all the vertices in an order such that no vertex appears before any other vertex that has a path to it
- ❖ Example input:



- ❖ Example output:
 - 126, 142, 143, 311, 331, 332, 312, 341, 351, 333, 352, 440

Activity: Valid Topological Sorts

- ❖ List 3 valid sorts:

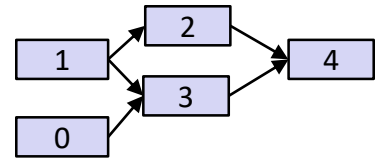


- ❖ Why do we perform topological sorts only on DAGs?
 - *A cycle means there is no correct answer*
- ❖ Does a DAG always have a unique answer?
 - *No; there can be 1 or more answers, depending on the graph*
- ❖ What DAGs have exactly 1 answer?
 - *A list*
- ❖ *Terminology:* A DAG represents a **partial order**, and a topological sort produces a **total order** that is consistent with it

Topological Sort: Applications

- ❖ Figuring out how to finish your degree
- ❖ Determining the order for recomputing spreadsheet cells
- ❖ Computing the order to compile files using a Makefile
- ❖ Scheduling jobs in a big data pipeline
- ❖ *Often: finding an order of execution for a dependency graph*

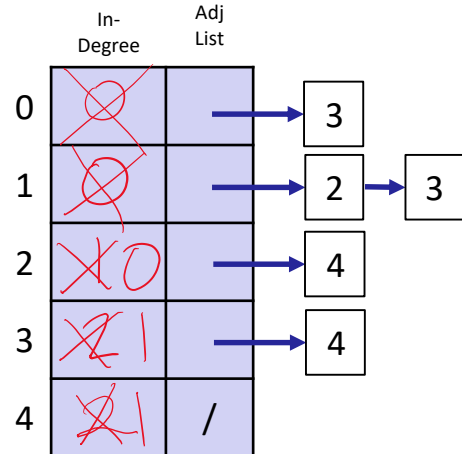
TopoSort: A Naïve Algorithm



- Label (“mark”) each vertex with its in-degree
 - Could write directly into a vertex’s field or a parallel data structure (e.g., array)
- While there are vertices not yet output:
 - Choose a vertex v with labeled with in-degree of 0
 - Output v and conceptually remove it from the graph
 - Foreach vertex w adjacent to v :
 - Decrement the in-degree of w

See also:
Detailed demo
on website

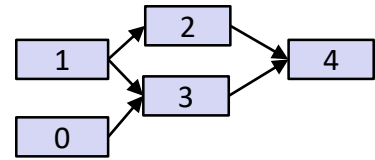
$\{0\} \{23\} 4$
 $\{10\} \{32\} 4$
 Any ordering of these
 pairs is OK



TopoSort: Notes

- ❖ Needed a vertex with in-degree of 0 to start
 - Remember: graph must be acyclic!
- ❖ If >1 vertex with in-degree=0, can break ties arbitrarily
 - Potentially many different correct orders!

Naïve TopoSort: Running Time?



$O(V+E)$

```

labelEachVertexWithItsInDegree();
for (i=0; i < numVertices; i++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
    
```

V times

$O(V)$
 $O(L)$

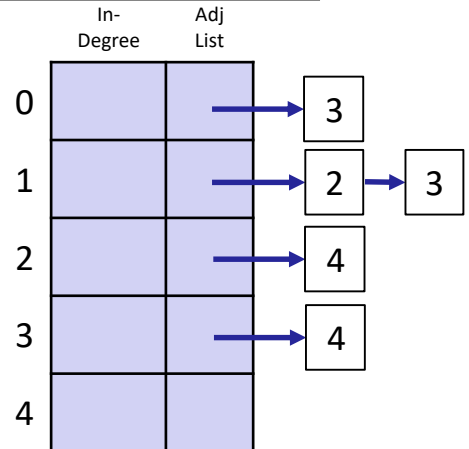
$O(V \times V)$, but $\Theta(E)$

Runtime: $(V+E) + V^2 + V + E$

$\in O(V^2 + E)$

But remember $E \leq V^2$, so:

$\in O(V^2)$

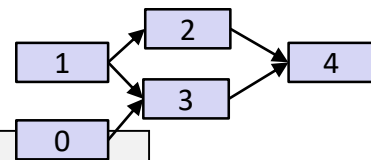


TopoSort's Runtime: Doing Better

- ❖ Avoid searching for a zero-degree node every time!
 - Keep the “pending” 0-degree nodes in a list, stack, queue, table, etc
 - The order we process them affects output, but not correctness or efficiency (*as long as add/remove are both $O(1)$*)

- ❖ Using a queue:
 - Label each vertex with its in-degree, enqueueing 0-degree nodes
 - While “pending” queue is not empty:
 - $v = \text{dequeue}()$
 - Output v and remove it from the graph
 - For each vertex w adjacent to v (i.e. w such that (v,w) in E):
 - decrement the in-degree of w
 - if new degree is 0, enqueue it

Better TopoSort: Running Time?



$O(V+E)$

```

labelAllAndEnqueueZeros();
for (i=0; i < numVertices; i++){
    v = dequeue();
    put v next in output
    for each w adjacent to v
        w.indegree--;
        if (w.indegree == 0)
            enqueue(w);
}
    
```

$O(1)$
 $O(1)$

V times

$O(V \times V)$, but $\Theta(E)$

Runtime: $(V+E) + V + V + E$
 $\in O(V + E)$

	In-Degree	Adj List
0		→ 3
1		→ 2 → 3
2		→ 4
3		→ 4
4		

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List

- ❖ Topological Sort

- ❖ **Traversals**
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - Conclusion

Lecture questions: pollev.com/cse332

Tree and Graph Reachability

- ❖ Find all nodes *reachable* from a starting node v
 - ie, there exists a path
 - Might “do something” at each visited node (an iterator!)
 - “Do something” is called *visiting* or *processing* a node
 - eg, print to output, set some field, etc.
 - *Traversing a node* or *iterating over a node* is different!
 - Just fetch adjacent/child nodes

- ❖ Related Questions:
 - Is an undirected graph connected?
 - Is a directed graph weakly / strongly connected?
 - For strongly, need a cycle back to starting node

Tree and Graph Traversals

- ❖ Can answer reachability with a *tree traversal* or *graph traversal*
 - Iterates over every node in a graph in some defined ordering
 - “Processes” or “visits” its contents
- ❖ There are several types of tree traversals
 - Level Order Traversal aka Breadth-First Traversal
 - Depth-First Traversal
 - Pre-order Traversal
 - In-order Traversal
 - Post-order Traversal

Tree/Graph Traversal: High-level Algorithm

❖ High-level Algorithm:

■ Initialization:

- Create an empty data structure to track “remaining work”
- Mark start as visited

■ While we still have work, follow the nodes:

- Get a node
- Visit/process that node
- Update its neighbors (eg, add to “remaining work” if it’s not already there)

```

traverseGraph(Node start) {
    pending = emptySet();
    pending.add(start);
    mark start as visited
}

while (!pending.empty()) {
    next = pending.remove();
    process(next);
    foreach u adjacent to next
        if (!u.marked)
            mark u
            pending.add(u)
}

```

Handwritten annotations:

- init* (bracketed around initialization)
- get more work* (pointing to `while`)
- has work* (pointing to `!pending.empty()`)
- do the thing* (pointing to `process(next)`)
- update neighbors* (bracketed around the inner loop)

❖ **Memorize this 5-step pattern!**

Tree/Graph Traversal: Running Time

- ❖ Assuming `add()` and `remove()` are $O(1)$, traversal is $O(|E|)$
 - Remember: we default to using an adjacency list

Tree/Graph Traversal: Order

- ❖ The order we process() depends *entirely* on how pending.add() and pending.remove() are implemented
 - Queue:
 - Tree: Level-order traversal
 - Graph: Breadth-first graph search (BFS)
 - Stack:
 - Tree: Depth-first search (3 flavors!)
 - Graph: Depth-first graph search (DFS)

- ❖ DFS and BFS are “big ideas” in computer science
 - Depth: explore one part before exploring other unexplored parts
 - Breadth: explore parts closer to the start before exploring farther parts

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List

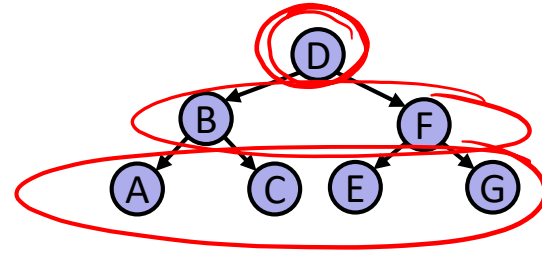
- ❖ Topological Sort

- ❖ Traversals
 - **Trees and Graphs: Level-order / Breadth-first**
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - Conclusion

Lecture questions: pollev.com/cse332

Trees: Level-Order

- ❖ Process top-to-bottom, left-to-right
 - Like reading in English
 - Goes “broad” instead of “deep”
- ❖ Resembles how we converted our binary heap (ie, a complete tree) to its array representation



D B F A C E G

Graphs: Breadth-First

- ❖ Note: we call the need-to-explore vertices “the fringe”

Our 5-step pattern!

```
BFS(Node start) {  
  q.enqueue(start)  
  mark start as visited
```

] init

```
  while (!q.empty())
```

has work

get work

do the thing

```
    next = q.dequeue()
```

```
    process(next)
```

```
    foreach u adjacent to next
```

```
      if (!u.marked)
```

```
        mark u
```

```
        q.enqueue(u)
```

] update neighbors

```
}
```

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List

- ❖ Topological Sort

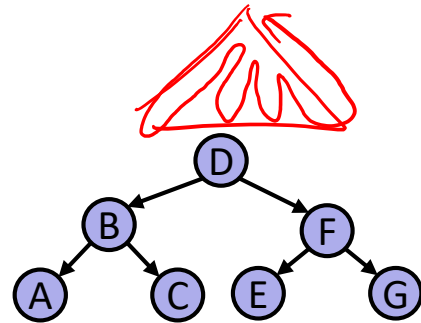
- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - **Trees: Three Flavors of Depth-first**
 - Graphs: Depth-first
 - Conclusion

Lecture questions: pollev.com/cse332

Trees: Depth-First Traversal

- ❖ Process deep nodes before shallow ones
 - Eg, visit A before F

- ❖ Comes in 3 different flavors

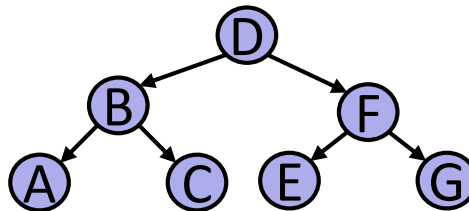


Trees: Depth-First: Pre-Order

- ❖ Pre-order “visits” the node before traversing its children

- DBACFEG

```
preOrder(Node x) {  
    if (x == null)  
        return;  
    → process(x.key)  
    preOrder(x.left)  
    preOrder(x.right)  
}
```

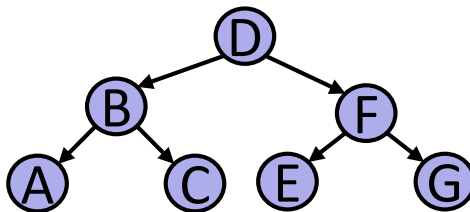


Trees: Depth-First: In-Order

- ❖ Pre-order “visits” the node before traversing its children
 - DBACFEG
- ❖ In-order traverses the left child, visits the node, then traverses the right child
 - ABCDEF

```
preOrder(Node x) {  
    if (x == null)  
        return;  
    process(x.key)  
    preOrder(x.left)  
    preOrder(x.right)  
}
```

```
inOrder(Node x) {  
    if (x == null)  
        return;  
    inOrder(x.left)  
    process(x.key)  
    inOrder(x.right)  
}
```



Trees: Depth-First: Post-Order

- ❖ Pre-order “visits” the node before traversing its children

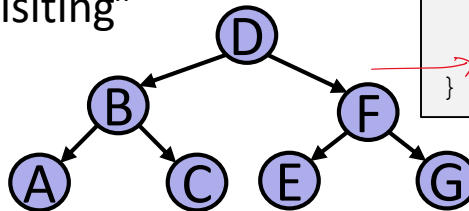
- DBACFEG

- ❖ In-order traverses the left child, “visits” the node, then traverses the right child

- ABCDEF

- ❖ Post-order traverses its children before “visiting” the node

- ACBEGFD



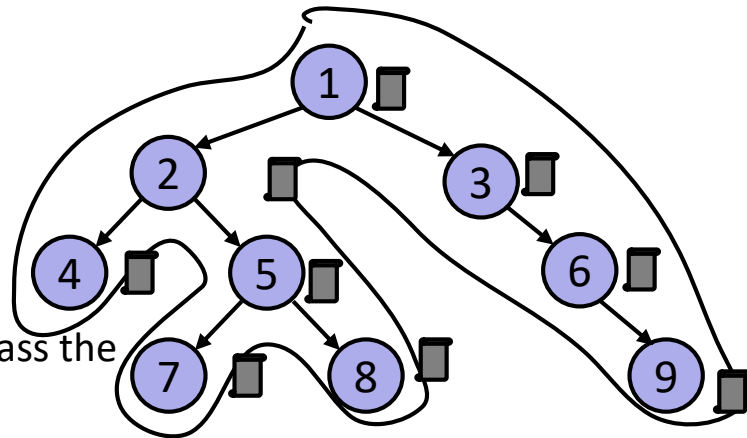
```
preOrder(Node x) {  
    if (x == null)  
        return;  
    process(x.key)  
    preOrder(x.left)  
    preOrder(x.right)  
}
```

```
inOrder(Node x) {  
    if (x == null)  
        return;  
    inOrder(x.left)  
    process(x.key)  
    inOrder(x.right)  
}
```

```
postOrder(Node x) {  
    if (x == null)  
        return;  
    postOrder(x.left)  
    postOrder(x.right)  
    process(x.key)  
}
```

Useful Trick for Depth-First Tree Traversals

- ❖ (Useful for humans, not algorithms)
- ❖ Trace a path around the graph, from the top going counter-clockwise
 - Pre-order: “Visit” when you pass the LEFT side of a node
 - In-order: “Visit” when you pass the BOTTOM of a node
 - Post-order: “Visit” when you pass the RIGHT side of a node.



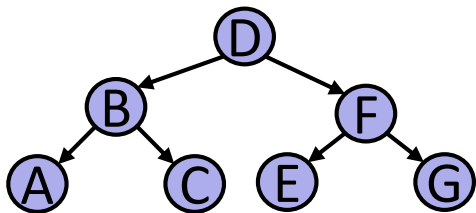
Example: post-order
4 7 8 5 2 9 6 3 1

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List
- ❖ Topological Sort
- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - **Graphs: Depth-first**
 - Conclusion

Lecture questions: pollev.com/cse332

Trees and Graphs: Depth-First



Our 5-step pattern!

```
DFSIterative(Node start) {  
  s.push(start)  
  mark start as visited  
  
  while (!s.empty())  
    next = s.pop()  
    process(next)  
    foreach u adjacent to next  
      if (!u.marked)  
        mark u  
        q.push(u)  
}
```

init

has work
get work
do the thing

update
neighbors

Lecture Outline

- ❖ Graph Representations, cont.
 - Review: Adjacency Matrix
 - Adjacency List

- ❖ Topological Sort

- ❖ Traversals
 - Trees and Graphs: Level-order / Breadth-first
 - Trees: Three Flavors of Depth-first
 - Graphs: Depth-first
 - **Conclusion**

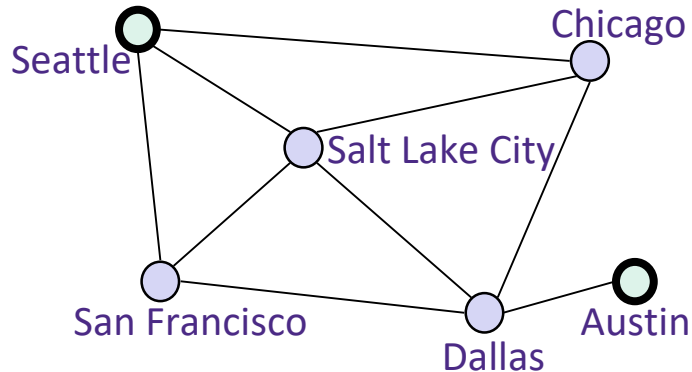
Lecture questions: pollev.com/cse332

Saving the Path

- ❖ These graph traversals can answer the “reachability question”:
 - “Is there a path from node x to node y ?”
- ❖ But what if we want to output the actual path or its length?
 - Eg, getting driving directions vs knowing it’s possible to get there
- ❖ Modifications:
 - Instead of just “marking” a node, store the path’s previous node
 - ie: when processing u , if we add v to the “remaining work” set $v.\text{prev}$ to u
 - When you reach the goal, follow `prev` fields backwards to start
 - (don’t forget to reverse the answer)
 - Path length:
 - Same idea, but also store integer distance at each node

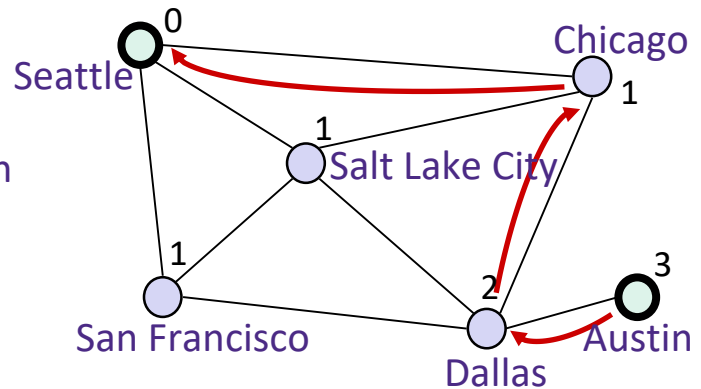
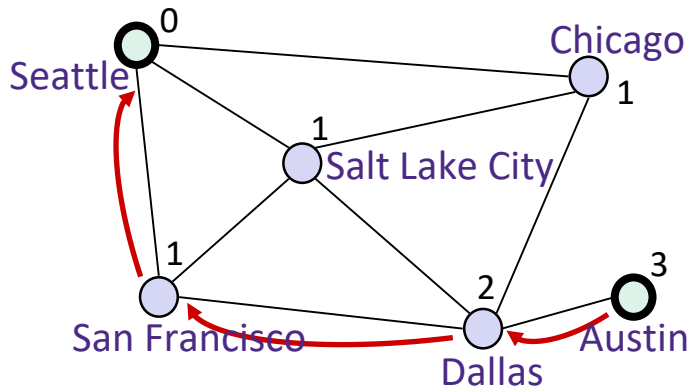
Saving the Path: Example using BFS (1 of 2)

- ❖ Find the shortest path from Seattle to Austin
 - Remember marked nodes are not re-enqueued
 - Shortest paths may not be unique



Saving the Path: Example using BFS (2 of 2)

- ❖ Find the shortest path from Seattle to Austin
 - Remember marked nodes are not re-enqueued
 - Shortest paths may not be unique



DFS/BFS Comparison

❖ Breadth-first search:

- Always finds shortest paths, i.e., finds “optimal solutions”
 - Better for “what is the shortest path from x to y?”
- But queue may hold up to $O(|V|)$ nodes
 - Eg, at the bottom level of perfect binary tree, queue contains $|V|/2$ nodes

❖ Depth-first search:

- Can use less space when finding a path
 - If longest path in the graph is p and highest out-degree is d then stack never has more than $d \cdot p$ elements

It Doesn't Have to be Either/Or

- ❖ A third approach: Iterative deepening (IDDFS):
 - Try DFS, but don't allow recursion more than K levels deep
 - If fails to find a solution, increment K and start the entire search over
- ❖ Like BFS, finds shortest paths. Like DFS, less space

Summary

- ❖ Two very different “standard” graph representations
 - Must understand tradeoffs to choose between adj list and adj matrix
- ❖ TopoSort finds a total ordering in a DAG representing a partial ordering
 - Runtime for TopoSort was dependent on graph representation and a helper data structure!
- ❖ We can traverse both trees and graphs
 - Depth-first-style tree traversals have 3 flavors (named by when the processing happens)
 - Breadth-first-style tree traversals are called “level-order”
 - Graphs can have “pre-” and “post-” style traversals, but ordering is less important than in trees