Today

• Announcements
• Questions?
• Graph Review
• Graph Traversals
  – Breadth First Search
  – Depth First Search
Announcements

• Midterm is Over!

• Project 2 Phase B:
  – Due Tuesday February 17\textsuperscript{th} 11pm

• Written HW 4 part B
  – Tonight 11pm

• Written HW 5 is out
  – Due Friday February 20\textsuperscript{th} 11pm
Questions

Questions about Written Homework 4b?

Questions about Project 2?

Other questions?
Graph Review

\[ G = (V, E) \]

- Contains set of vertices and set of edges
  - \(|V| = \) number of vertices
  - \(|E| = \) number of edges

- Max \(|E|\) for undirected graph
  \[ |V| + (|V| - 1) + (|V| - 2) + ... + 1 = |V|(|V| + 1) / 2 \]

- Max \(|E|\) for directed graph
  \[ |V| + |V| + |V| + ... + |V| = |V|^* |V| = |V|^2 \]
Graph Review

Path

• List of vertices \([v_0, v_1, ..., v_n]\), such that 
  \((v_i, v_{i+1}) \in E\) for all \(0 \leq i < n\)
  – Path length = number of edges on path
  – Path cost = sum of all edge weights on path

Cycle

• A path that begins and ends at the same node
Graph Review: Undirected

• Edges have no directions
• Connected
  – There is a path between all pairs of vertices
• Fully Connected
  – There is an edge between all pairs of vertices
Graph Review: Directed

• Edges have direction
• Weakly Connected
  – There is an undirected path between all pairs of vertices
• Strongly Connected
  – There is a directed path between all pairs of vertices
• Fully Connected
  – If there is edge (both way) between all pairs of vertices
Graph Traversals

• For an arbitrary graph and a starting node $v$, find all nodes reachable (i.e., there exists a path) from $v$
  – Possibly “do something” for each node (an iterator!)
    • E.g. Print to output, set some field, etc.

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

• Basic idea:
  – Keep following nodes
  – But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once
Graph Traversals: Abstract Idea

traverseGraph(Node start) {
    Set pending = emptySet();
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove();
        for each node u adjacent to next
            if(u is not marked) {
                mark u
                pending.add(u)
            }
    }
}


Graph Traversals: Running Time

• Assuming add and remove are O(1), entire traversal is O(|E|)
  – Use an adjacency list representation

• The order we traverse depends entirely on how add and remove work/are implemented
  – Depth-first graph search (DFS): a stack
  – Breadth-first graph search (BFS): a queue

• DFS and BFS are “big ideas” in computer science
  – Depth: recursively explore one part before going back to the other parts not yet explored
  – Breadth: Explore areas closer to the start node first
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end

• BFS starting from node X

Order Processed:

\[
\begin{array}{c}
X \\
Y \\
Z \\
W \\
R \\
\end{array}
\]

Mark X, and enqueue it

\[
\begin{array}{c|c}
& X \\
\hline
\end{array}
\]
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end
• BFS starting from node X

Order Processed: X

Dequeue X, process it, mark and enqueue X’s neighbors

| Z | Y |
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end
• BFS starting from node X

Order Processed: X Y

Dequeue Y, process it, mark and enqueue Y’s neighbors

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>Z</th>
</tr>
</thead>
</table>
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end

• BFS starting from node X

Order Processed: X Y Z

Note: Do not add neighbors that have already been marked

Dequeue Z, process it, mark and enqueue Z’s neighbors

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>R</th>
</tr>
</thead>
</table>

X -> Y -> R -> W -> Z
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end

• BFS starting from node X

Order Processed: X Y Z R

Dequeue R, process it, mark and enqueue R’s neighbors
Breadth First Search

• Pick the shallowest unmarked node
  – Uses a **queue**, enqueue new nodes at the end

• BFS starting from node X

Order Processed: X Y Z R W

Dequeue W, process it, mark and enqueue W’s neighbors

NOTE: Do not add neighbors that have already been marked
Breadth First Search

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

• Order Processed: A B C D E F G H
  – A “level-order” traversal
Depth First Search

• Pick the deepest unmarked node
  – Uses a stack, push new nodes on the top

• DFS starting from node X

Order Processed:

Mark X and push it
Depth First Search

• Pick the deepest unmarked node
  – Uses a stack, push new nodes on the top

• DFS starting from node X

Order Processed: X

Pop X, process it, mark and push X’s neighbors
Depth First Search

- Pick the deepest unmarked node
  - Uses a **stack**, push new nodes on the top
- DFS starting from node X

Order Processed: X Z

![Diagram](image)

Pop Z, process it, mark and push Z’s neighbors

NOTE: Do not add neighbors that are already marked
Depth First Search

• Pick the deepest unmarked node
  – Uses a stack, push new nodes on the top

• DFS starting from node X

Order Processed: X Z W

Pop W, process it, mark and push W’s neighbors
Depth First Search

• Pick the deepest unmarked node
  – Uses a **stack**, push new nodes on the top
• DFS starting from node X

Order Processed: X Z W R

Pop R, process it, mark and push R’s neighbors
Depth First Search

• Pick the deepest unmarked node
  – Uses a **stack**, push new nodes on the top
• DFS starting from node X

Order Processed: X Z W R Y

![Graph](image)

Pop Y, process it, mark and push Y’s neighbors

NOTE: Do not add neighbors that are already marked
Depth First Search: Recursive

DFS(Node start) {
    mark and “process” (e.g. print) start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}

Order Processed: A B D E C F G H
• Exactly what we called a “pre-order traversal” for trees
• The marking is not needed here, but we need it to support arbitrary graphs, we need a way to process each node exactly once
Depth First Search: With a Stack

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order Processed: A C F H G B E D
• A different but perfectly fine traversal
DFS/BFS Comparison

Breadth-first search:
• Always finds shortest paths, i.e., “optimal solutions”
  – Better for “what is the shortest path from $x$ to $y$”
• Queue may hold $O(|V|)$ nodes (e.g. at the bottom level of binary tree of height $h$, $2^h$ nodes in queue)

Depth-first search:
• Can use less space in finding a path
  – If longest path in the graph is $p$ and highest out-degree is $d$ then DFS stack never has more than $d*p$ elements

A third approach: *Iterative deepening (IDDFS):*
• Try DFS but don’t allow recursion more than $K$ levels deep.
  – If that fails, increment $K$ and start the entire search over
• Like BFS, finds shortest paths. Like DFS, less space.
Saving the Path

• Our graph traversals can answer the “reachability question”:
  – “Is there a path from node x to node y?”
• Q: But what if we want to output the actual path?
  – Like getting driving directions rather than just knowing it’s possible to get there!
• A: Like this:
  – Instead of just “marking” a node, store the previous node along the path (when processing u causes us to add v to the search, set v.path field to be u)
  – When you reach the goal, follow path fields backwards to where you started (and then reverse the answer)
  – If just wanted path length, could put the integer distance at each node instead