Strongly Connected Components
and Breadth-First Search
CSE 101: Design and Analysis of Algorithms
Lecture 4
CSE 101: Design and analysis of algorithms

• Strongly connected components and breadth-first search
  – Reading: Sections 3.4, 4.1, 4.2, and 4.3

• Homework 1 due today, 11:59 PM

• Homework 2 will be assigned today
  – Due Oct 16, 11:59 PM
Depth-first search on directed graphs

Based on slides courtesy of Miles Jones
Depth-first search on directed graphs (reverse alphabetical order)
Edge types (directed graph)

- Tree edge: solid edge included in the depth-first search output tree
- Back edge: leads to an ancestor
- Forward edge: leads to a descendent
- Cross edge: leads to neither ancestor or descendent

- Note that back edge is slightly different in directed and undirected graphs
Edge types and pre/post numbers

• The different types of edges can be determined from the pre/post numbers for the edge \((u, v)\)
  – \((u, v)\) is a tree/forward edge then
    \(pre(u) < pre(v) < post(v) < post(u)\)
  – \((u, v)\) is a back edge then
    \(pre(v) < pre(u) < post(u) < post(v)\)
  – \((u, v)\) is a cross edge then
    \(pre(v) < post(v) < pre(u) < post(u)\)
Cycles in directed graphs

• A cycle in a directed graph is a path that starts and ends with the same vertex

\[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0 \]

\[ A \rightarrow C \rightarrow E \rightarrow A \]
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: →
  – Suppose G has a cycle

\[ v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0 \]
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: →
  – Suppose G has a cycle
    \[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0 \]
  – Suppose \( v_i \) is the first vertex to be discovered
    • What does that mean about \( v_i \)?
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: →
  – Suppose G has a cycle
    \[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0 \]
  – Suppose \( v_i \) is the first vertex to be discovered
    • What does that mean about \( v_i \)?
      – It is the vertex with the lowest pre number
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: →
  – Suppose G has a cycle
    \[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0 \]
  – Suppose \( v_i \) is the first vertex to be discovered
    • All other \( v_j \) are reachable from it and therefore, they are all descendants in the depth-first search tree
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: →
  – Suppose G has a cycle
    \[ v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0 \]
  – Suppose \( v_i \) is the first vertex to be discovered
    • All other \( v_j \) are reachable from it and therefore, they are all descendants in the depth-first search tree
  – Therefore the edge \((v_{i-1}, v_i)\) is a back edge ■
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: ←
  – Suppose $(u, v)$ is a back edge
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: ⪞
  – Suppose \((u, v)\) is a back edge
    • \(u\) is a descendant of \(v\) in the depth-first search output tree
    • There is a path from \(v\) to \(u\)
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: ←
  – Suppose \((u, v)\) is a back edge
  – Then by definition \(v\) is a ancestor of \(u\) so there is a path from \(v\) to \(u\) in the depth-first search output tree
A directed graph has a cycle if and only if its depth-first search reveals a back edge

• Proof: ←
  – Suppose \((u, v)\) is a back edge
  – Then by definition \(v\) is a ancestor of \(u\) so there is a path from \(v\) to \(u\) in the depth-first search output tree
  – Along with the back edge, this path completes a cycle □
Directed acyclic graphs (dags)

• A directed graph without a cycle is called **acyclic**
• We can test whether a graph is acyclic with depth-first search
  – Step 1: perform depth-first search on the graph
    • Keep track of pre and post numbers
  – Step 2: test each edge \((u, v)\) to see if it is a back edge
    • If \(post(u) < post(v)\) (i.e., \((u, v)\) is back edge), then return false
Linearization of dags

- Is it possible to order the vertices such that all edges go in only one direction?
- For what types of dags is this possible?
- How do we find such an ordering?
Directed acyclic graphs (dags)

• Property: every edge in a dag goes from a higher post number to lower post number

• Proof: suppose $(u, v)$ is an edge in a dag, then it cannot be a back edge. Therefore it can only be a forward edge/tree edge or a cross edge.
  – Both of which have the property that $post(v) < post(u)$
Edge types and pre/post numbers

- The different types of edges can be determined from the pre/post numbers for the edge \((u, v)\)
  - \((u, v)\) is a tree/forward edge then
    \[\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)\]
  - \((u, v)\) is a back edge then
    \[\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)\]
  - \((u, v)\) is a cross edge then
    \[\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)\]
Linearization of a dag

- Since we know that edges go in the direction of decreasing post numbers, if we order the vertices by decreasing post numbers then we will have a linearization

```plaintext
procedure linearize(a dag G=(V,E))
  run DFS(G)
  return list of vertices in decreasing order of post numbers.
```
Linearization of a dag

procedure linearize(a dag G=(V,E))
  run DFS(G)  Keep track of pre and post numbers $O(n + m)$
  return list of vertices in decreasing order of post numbers. $O(n \log n)$

Total time: $O(n + m + n \log n)$

Alternatively, add vertices to an output list as you assign post numbers, then reverse list $O(n + m)$
Sources and sinks

• Since all dags can be linearized that means the first vertex in the ordering does not have any edges coming in and the last vertex does not have any edges going out

• Definitions
  – A vertex with no incoming edges is called a source
  – A vertex with no outgoing edges is called a sink

• Property: all dags have at least one source and one sink
Pre and post numbers in dags

• The vertex with the highest post number is always?

• The vertex with the lowest post number is always?
Pre and post numbers in dags

• The vertex with the highest post number is always?
  – Source

• The vertex with the lowest post number is always?
  – Sink
Example
Example
Strongly connected vertices

- Two vertices $u$ and $v$ in a directed graph are strongly connected if there exists a path from $u$ to $v$ and a path from $v$ to $u$
- Which vertices are strongly connected to $J$?
Strongly connected vertices

- Two vertices $u$ and $v$ in a directed graph are strongly connected if there exists a path from $u$ to $v$ and a path from $v$ to $u$
- Which vertices are strongly connected to J?
  - K-L-M-J
Strongly connected graph

• A graph is called strongly connected if for each pair of vertices v, u there is a path from v to u and a path from u to v

• Is this a strongly connected graph?
Strongly connected graph

- A graph is called strongly connected if for each pair of vertices v, u there is a path from v to u and a path from u to v
- Is this a strongly connected graph?
  - No
Strongly connected components

• Consider the relation $uRv$ if $u$ is strongly connected to $v$
• Then $R$ is an equivalence relation. It is reflexive, symmetric, and transitive.
• So $R$ partitions $V$, the set of vertices into equivalence classes
• These equivalence classes are called strongly connected components
Strongly connected components

• What are the strongly connected components of this graph?
Strongly connected components

• What are the strongly connected components of this graph?
Strongly connected components as vertices
Strongly connected components

• Property: every directed graph is a dag of its strongly connected components
• Some strongly connected components are sinks and some are sources
Strongly connected components as vertices
Decomposition of a directed graph

- There is a linear time algorithm that decomposes a directed graph into its strongly connected components.
- If explore is performed on a vertex $u$, then it will visit only the vertices that are reachable by $u$.
- What vertices will be visited when explore is performed on $u$ if it is in a sink strongly connected component?
Decomposition of a directed graph

- There is a linear time algorithm that decomposes a directed graph into its strongly connected components.
- If explore is performed on a vertex \( u \), then it will visit only the vertices that are reachable by \( u \).
- What vertices will be visited when explore is performed on \( u \) if it is in a sink strongly connected component?
  - Only the vertices in that strongly connected component will be reached.
Sink strongly connected components

- If explore is performed on a vertex that is in a sink strongly connected component, then only the vertices from that strongly connected component will be visited.
- This suggests a way to look for strongly connected components:
  - Start explore on a vertex in a sink strongly connected component and visit its strongly connected component.
  - Remove the sink strongly connected component from the graph and repeat.
Source strongly connected components

• Ideally, we would like to find a vertex in a sink strongly connected component
  – There is not a direct way to do this
• However, there is a way to find a vertex in a source strongly connected component
  – The vertex with the highest post number in any depth-first search output tree belongs to a source strongly connected component
  • Note: the vertex with the lowest post number in a depth-first search output does not necessarily belong to a sink strongly connected component
Example where lowest post number is not in a sink
Example where lowest post number is not in a sink
Source strongly connected components

• The vertex with the highest post number in any depth-first search output tree belongs to a source strongly connected component

• To prove this, we will state a more general property
  – If $C$ and $C'$ are strongly connected components and there is an edge from a vertex in $C$ to a vertex in $C'$ then the highest post number in $C$ is greater than the highest post number in $C'$
Proof

• Case 1: depth-first search searches $C$ before $C'$:
  – At some point depth-first search will cross into $C'$ and visit every edge in $C'$, then it will retrace its steps until it gets back to the first node in $C$ it started with and assign it the highest post number

![Diagram showing nodes C and C']
Proof

• Case 2: depth-first search searches $C'$ before $C$:
  – Depth-first search will visit all vertices of $C'$ before getting stuck and assign a post number to all vertices of $C'$, then it will visit some vertex of $C$ later and assign post numbers to those vertices
Corollary

• The strongly connected components can be linearized by arranging them in decreasing order of their highest post numbers
Finding sink strongly connected components

• Given a graph $G$, let $G^R$ be the reverse graph of $G$
  – Then, the sources of $G^R$ are the sinks of $G$
• If we perform depth-first search on $G^R$, then the vertex with the highest post number is in a source. This means that this vertex will be in a sink of $G$.
• Start with this vertex and explore the strongly connected component
• Then, the vertex with the next highest post number in $G^R$ is in the next strongly connected component in linear order, so start with that one next
Decomposing a graph into its strongly connected components

- Generate $G^R$
- Run depth-first search on $G^R$, keeping track of the post numbers
- Run depth-first search on $G$ and order the vertices in decreasing order of the post numbers from the previous step. Every time depth-first search increments cc, you have found a new strongly connected component.
Example
Example
Decomposing a graph into its strongly connected components

- Generate $G^R$
- Run depth-first search on $G^R$, keeping track of the post numbers
- Run depth-first search on $G$ and order the vertices in decreasing order of the post numbers from the previous step. Every time depth-first search increments $cc$, you have found a new strongly connected component.

- How long does this take?
Decomposing a graph into its strongly connected components

- Generate $G^R \ O(n + m)$
- Run depth-first search on $G^R$, keeping track of the post numbers $O(n + m)$
- Run depth-first search on $G$ and order the vertices in decreasing order of the post numbers from the previous step. Every time depth-first search increments cc, you have found a new strongly connected component. $O(n + m)$

- How long does this take? $O(n + m)$
Depth-first search is good for

- Find what vertices can be reached by a given vertex
- Divide an undirected graph into connected components
- Find cycles in graphs (directed or undirected)
- Find sinks and sources in dags
- Topologically sort a dag
- Make a directed graph into a dag of its strongly connected components
Depth-first search is not good for

- Finding shortest distances between vertices
Next lecture

• Dijkstra’s algorithm and priority queue implementations
  – Reading: Sections 4.4 and 4.5