Graph Algorithms Using Depth First Search

Analysis of Algorithms
Week 8, Lecture 1

Prepared by
John Reif, Ph.D.
Distinguished Professor of Computer Science
Duke University
Graph Algorithms Using Depth First Search

a) Graph Definitions
b) DFS of Graphs
c) Biconnected Components
d) DFS of Digraphs
e) Strongly Connected Components
Readings on Graph Algorithms
Using Depth First Search

• Reading Selection:
  – CLR, Chapter 22
Graph Terminology

- **Graph**: \( G=(V,E) \)
- **Vertex set**: \( V \)
- **Edge set**: \( E \) pairs of vertices which are adjacent
Directed and Undirected Graphs

- $G$ directed
  
  if edges ordered pairs $(u,v)$
  
  $u \rightarrow v$

- $G$ undirected
  
  if edges unordered pairs $\{u,v\}$
  
  $u \leftrightarrow v$
Proper Graphs and Subgraphs

- **Proper graph:**
  - No loops
  - No multi-edges

- **Subgraph** $G'$ of $G$
  
  $G' = (V', E')$ where
  
  $V'$ is a subset of $V$ and $E'$ is a subset of $E$
Paths in Graphs

- **Path** \( p \)

\[ V_0 \quad e_1 \quad V_1 \quad e_2 \quad V_2 \quad \ldots \quad e_k \quad V_{k-1} \quad V_k \]

\( P \) is a sequence of vertices \( v_0, v_1, \ldots, v_k \)

where for \( i = 1, \ldots, k \), \( v_{i-1} \) is adjacent to \( v_i \)

Equivalently, \( p \) is a sequence of edges \( e_1, \ldots, e_k \)

where for \( i = 2, \ldots, k \) each consecutive pair of edges \( e_{i-1}, e_i \) share a vertex
Simple Paths and Cycles

- **Simple path**
  no edge or vertex repeated, except possibly $v_o = v_k$
- **Cycle**
  a path $p$ with $v_o = v_k$ where $k > 1$
A Connected Undirected Graph

G is *connected* if $\exists$ path between each pair of vertices
else $G$ has $\geq 2$ connected components:
maximal connected subgraphs
Biconnected Undirected Graphs

G is \textit{biconnected} if \exists two disjoint paths between each pair of vertices

(or G is single edge)
Size of a Graph

- **Graph** $G = (V,E)$
  - $n = |V| = \# \text{ vertices}$
  - $m = |E| = \# \text{ edges}$
  - size of $G$ is $n+m$
Representing a Graph by an Adjacency Matrix

- **Adjacency Matrix** $A$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*space cost $n^2 - n$*
Adjacency List Representation of a Graph

- **Adjacency Lists** $\text{Adj (1)}, \ldots, \text{Adj (n)}$
  $\text{Adj}(v)$ = list of vertices adjacent to $v$

Space cost $O(n+m)$
Definition of an Undirected Tree

- **Tree**
  
  T is graph with unique simple path between every pair of vertices

  \[ n = \# \text{ vertices} \]
  \[ n-1 = \# \text{ edges} \]

- **Forest**
  - set of trees
Definition of a Directed Tree

• **Directed Tree**
  T is digraph with distinguished vertex root \( r \) such that each vertex reachable from \( r \) by a unique path

*Family Relationships:*
  - ancestors
  - descendants
  - parent
  - child
  - siblings

*leaves have no proper descendants*
An Ordered Tree

- **Ordered Tree**
  - is a directed tree with siblings ordered
Preorder Tree Traversal

- **Preorder:** A,B,C,D,E,F,H,I
  - [1] root (order vertices as pushed on stack)
  - [2] preorder left subtree
  - [3] preorder right subtree

```
A
  B
  C
    D
    E
    F
    H
    I
```
Postorder Tree Traversal

- Postorder: B,E,D,H,I,F,C,A
  - [1] postorder left subtree
  - [2] postorder right subtree
  - [3] root (order vertices as popped off stack)
Spanning Tree and Forest of a Graph

- T is a *spanning tree* of graph G if
  1. T is a directed tree with the same vertex set as G
  2. each edge of T is a directed version of an edge of G

- *Spanning Forest:*
  forest of spanning trees of connected components of G
Example Spanning Tree of a Graph

- root
- tree edge
- back edge
- No cross edge
Classification of Edges of G with Spanning Tree T

- An edge \((u,v)\) of T is **tree edge**

- An edge \((u,v)\) of G-T is **back edge** if \(u\) is a descendent or ancestor of \(v\).

- Else \((u,v)\) is a **cross edge** *(do not exist in undirected DFS)*
Tarjan’s Depth First Search Algorithm

- We assume a Random Access Machine (RAM) computational model
- Algorithm Depth First Search

**Input** graph $G = (V,E)$ represented by adjacency lists $\text{Adj}(v)$ for each $v \in V$

1. $N \leftarrow 0$
2. for all $v \in V$ do (number $(v) \leftarrow 0$
   - $\text{children } (v) \leftarrow ()$ end
3. for all $v \in V$ do
   - if number $(v) = 0$ then $\text{DFS}(v)$
4. output spanning forest defined by children
Recursive DFS Procedure

**recursive procedure**  
DFS(v)  

[1]  
N $\leftarrow$ N + 1; number (v) $\leftarrow$ N  

[2]  
for each $u \in \text{Adj}(v)$  

if number (u) = 0 then  

(add u to children (v); DFS(u))  

The preorder numbers give the order the vertices are first visited in DFS.
Time Cost of Depth First Search (DFS) Algorithm on a RAM

• Input size $n = |V|$, $m = |E|$
• Theorem Depth First Search takes linear time cost $O(n+m)$
• Proof
  Can associate with each edge and vertex
  a constant number of operations.
Classification of Edges of $G$ via DFS
Spanning Tree $T$

- *Edge notation induced by*
- *Depth First Search Tree $T$*

\[
\begin{align*}
    u \rightarrow v & \iff (u,v) \text{ is tree edge of } T \\
    u \rightarrow v & \iff u \text{ is an ancestor of } v \\
    u \rightarrow\leftarrow v & \iff (u,v) \text{ is back edge if } (u,v) \in G-T \\
    \text{with either } u \rightarrow v & \text{ or } v \rightarrow u
\end{align*}
\]
Classification of Edges of Graph $G$ via DFS Spanning Tree $T$

Preorder numbering vertices by order visited in DFS
Classification of Edges of $G$ via DFS
Spanning Tree $T$ (cont’ d)

- Note DFS tree $T$ of an undirected graph has no cross edges $(u,v)$ where $u,v$ are unrelated in $T$
Verifying Vertices are Descendants via Preordering of Tree

• Suppose we **preorder** number a Tree $T$
  Let $D_v = \# \text{ of descendants of } v \quad \text{(found in DFS)}$

• **Lemma**
  $u$ is descendant of $v$\iff $v \leq u < v + D_v$
Testing for Proper Ancestors via Preordering

- **Lemma**
  If \( u \) is descendant of \( v \) and \((u,w)\) is back edge s.t. \( w < v \) then \( w \) is a proper ancestor of \( v \)
Low Values

• For each vertex v,
  \[ \text{define } \text{low}(v) = \min ( \{v\} \cup \{w \mid v \rightarrow \ldots \rightarrow w\} ) \]

• Can prove by induction:
  \[ \text{Lemma} \]
  \[ \text{low}(v) = \min ( \{v\} \cup \{\text{low}(w) \mid v \rightarrow w\} \cup \{w \mid v \rightarrow \ldots \rightarrow w\} ) \]

  Can be computed during DFS in postorder.
Graph with DFS Numbering of vertices [Low Values in Brackets]
Biconnected Components

- $G$ is **Biconnected** iff either
  1. $G$ is a single edge, or
  2. for each triple of vertices $u,v,w$

    $\exists$ w-avoiding path from $u$ to $v$

    (equivalently: $\exists$ two disjoint paths from $u$ to $v$)
Example of Biconnected Components of Graph G

• Maximal subgraphs of G which are biconnected.
Biconnected Components Meet at Articulation Points

- The intersection of two biconnected components consists of at most one vertex called an Articulation Point.
- Example: 1, 2, 5 are articulation points
Discovery of Biconnected Components via Articulation Points

=> If can find articulation points then can compute biconnected components:

**Algorithm:**
- During DFS, use auxiliary stack to store visited edges.
- Each time we complete the DFS of a tree child of an articulation point, pop all stacked edges currently in stack.
- These popped off edges form a biconnected component.
Characterization of an Articulation Point

- **Theorem**
  
a is an *articulation point* iff either

  1. a is root with \( \geq 2 \) tree children
  
  or
  
  2. a is not root but a has a tree child v with
  
  low \((v) \geq a\)

  *(note easy to check given low computation)*
Proof of Characterization of an Articulation Point

- **Proof**
  The conditions are *sufficient* since any $a$-avoiding path from $v$ remains in the subtree $T_v$ rooted at $v$, if $v$ is a child of $a$

- To show condition *necessary*, assume $a$ is an articulation point.
Characterization of an Articulation Point (cont’d)

• **Case (1)**
  If \( a \) is a root and is articulation point, \( a \) must have \( \geq 2 \) tree edges to two distinct biconnected components.

• **Case (2)**
  If \( a \) is not root, consider graph \( G \) - \( \{a\} \) which must have a connected component \( C \) consisting of only descendants of \( a \), and with no backedge from \( C \) to an ancestor of \( v \). Hence \( a \) has a tree child \( v \) in \( C \) and \( \text{low}(v) \geq a \)
Proof of Characterization of an Articulation Point (cont’ d)

- Case (2)

Articulation Point $a$ is *not* the root

subtree $T_v$
Computing Biconnected Components

- **Theorem**
  The Biconnected Components of \( G = (V,E) \) can be computed in time \( O(|V| + |E|) \) using a RAM.

- **Proof idea**
  - Use characterization of Biconnected components via articulation points.
  - Identify these articulation points dynamically during depth first search.
  - Use a secondary stack to store the edges of the biconnected components as they are visited.
  - When an articulation point is discovered, pop the edges of this stack off to output a biconnected component.
Biconnected Components Algorithm

[0] initialize a STACK to empty
During a DFS traversal do
[1] add visited edge to STACK
[3] test if v is an articulation point
[4] if so, for each u ∈ children (v) in order
    where low (u) ≥ v
do    Pop all edges in STACK
upto and including tree edge (v,u)
Output popped edges as a
biconnected component of G
don
Time Bounds of Biconnected Components Algorithm

- *Time Bounds:*
  Each edge and vertex can be associated with $O(1)$ operations. So time $O(|V| + |E|)$. 
Depth First Search in a Directed Graph

Depth first search tree $T$ of Directed Graph

$G = (V,E)$

edge set $E$ partitioned:

The preorder numbers give order vertices are first visited in DFS:

$$i = \text{DFS number} = \text{order discovered}$$

The postorder numbers give order vertices are last visited in DFS:

$$j = \text{postorder} = \text{order vertex popped off DFS stack}$$
Classification of Directed Edge \((u, v)\) via DFS Spanning Tree \(T\)

- **Tree edge** \((u, v)\): \[\text{if } u \rightarrow v \text{ in } T \] \((u \text{ parent of } v \text{ in } T)\)

- **Cycle edge** \((u, v)\): \[\text{if } v \rightarrow u \] \((u \text{ is descendant of } v \text{ in } T)\)

- **Forward edge** \((u, v)\): \[\text{if } (u, v) \notin T \] \((u \text{ is descendant of } v \text{ in } T)\)

- **Cross edge** \((u, v)\): \[\text{otherwise} \] \((u, v \text{ not related in } T)\)

\[\ast\]
Topological Order of Acyclic Directed Graph

- Digraph $G = (V,E)$ is **acyclic** if it has no cycles

\[
\text{Topological Order} \\
V = \{v_1, \ldots, v_n\} \text{ satisfies} \\
(v_i, v_j) \in E \Rightarrow i < j
\]
Characterization of an Acyclic Digraph

$G$ is acyclic iff $\exists$ no cycle edge

- **Proof**
  
  Case (1):
  - Suppose $(u,v) \in E$ is a cycle edge, so $v \not\rightarrow u$.
  - But let $e_1,...,e_k$ be the tree edges from $v$ to $u$.
  - Then $(u,v), e_1,...,e_k$ is a cycle.
Characterization of an Acyclic Digraph (cont’ d)

Case (2):
• Suppose G has no a cycle edge
• Then order vertices in postorder of DFS spanning forest (i.e. in order vertices are popped off DFS stack).
• This is a reverse topological order of G.
• So, G can have no cycles.

Note:
Gives an $O(|V| + |E|)$ algorithm for computing Topological Ordering of an acyclic graph $G = (V,E)$.
Strong Components of Directed Graph

• Strong Component
  maximum set vertices
  $S$ of $V$ such that $\forall u, v \in S$ $\exists$ cycle containing $u, v$

• Collapsed Graph
  $G^*$ derived by collapsing each strong component into a single vertex.

note: $G^*$ is acyclic.
Algorithm Strong Components

Input digraph $G$


[2] Let $G^{-}$ be the reverse digraph derived from $G$ by reversing direction of each edge.
Algorithm Strong Components (cont’d)

Output resulting DFS tree of $G^-$ as a strongly connected component of $G$.

[4] repeat [3], starting at highest numbered vertex not so for visited (halt when all vertices visited)
Time Bounds of Algorithm Strong Components

- **Time Bounds**
  - each DFS costs time $c(|V| + |E|)$
  - So total time $= 2\cdot c(|V| + |E|) \\ \leq O(|V| + |E|)$
Example of Step [1] of Strong Components Algorithm

Example Input Digraph G
Example of Step [2] of Strong Components Algorithm

Reverse Digraph $G^-$
Proof of Strong Components Algorithm

- **Theorem**
  The Algorithm outputs the strong components of $G$.

- **Proof**
  We must show these are exactly the vertices in each DFS spanning forest of $G^\rightarrow$. 
Proof of Strong Components Algorithm (continued)

• Suppose:
  • v,w in the same strong component and DFS search in $G^-$ starts at vertex r and reaches v.
  
  Then w will also be reached in the DFS search in $G^-$. 

• So v,w are output together in the same spanning tree of $G^-$. 
Proof of Strong Components Algorithm (continued)

- Suppose:
  - $v,w$ output in same spanning tree of $G^-$.  

- Let $r$ be the root of that spanning tree of $G^-$.  

- Then there exists paths in $G^-$ from $r$ to each of $v$ and $w$.  

- So there exists paths in $G$ to $r$ from each of $v$ and $w$.  

Proof of Strong Components

Algorithm (continued)

• r is root of spanning tree of G containing v and w.

• Suppose: no path in G to r from v.

• Then since r has a higher postorder than v, there is no path in G from v to r, a contradiction.

• Hence, there exists path in G from r to v, and similar argument gives path from r to w.

• So v and w are in a cycle of G and must be in the same strong component, completing proof!
Graph Algorithms Using Depth First Search

Analysis of Algorithms
Week 8, Lecture 1

Prepared by
John Reif, Ph.D.
Distinguished Professor of Computer Science
Duke University