Graph Algorithms

- Many problems are naturally represented as graphs
  - Networks, Maps, Possible paths, Resource Flow, etc.
- Ch. 3 focuses on algorithms to find connectivity in graphs
- Ch. 4 focuses on algorithms to find paths within graphs
- $G = (V,E)$
  - $V$ is a set of vertices (nodes)
  - $E$ is a set of edges between vertices, can be directed or undirected
    - Undirected: $e = \{x,y\}$
    - Directed: $e = (x,y)$
  - Degree of a node is the number of impinging edges
    - Nodes in a directed graph have an in-degree and an out-degree
- WWW is a graph
  - Directed or undirected?
Graph Representation – Adjacency Matrix

- \( n = |V| \) for vertices \( v_1, v_2, \ldots, v_n \)
- Adjacency Matrix \( A \) is \( n \times n \) with
  \[
  a_{ij} = \begin{cases} 
  1 & \text{if there is an edge from } v_i \text{ to } v_j \\
  0 & \text{otherwise}
  \end{cases}
  \]
- \( A \) is symmetric if it is undirected
- One step lookup to see if an edge exists between nodes
- \( n^2 \) size is wasteful if \( A \) is sparse (i.e. not highly connected)
- If densely connected then \( |E| \approx |V|^2 \)
Graph Representation – Adjacency List

- Adjacency List: $n$ lists, one for each vertex
- Linked list for a vertex $u$ contains the vertices to which $u$ has an outgoing edge
  - For directed graphs each edge appears in just one list
  - For undirected graph each edge appears in both vertex lists
- Size is $O(|V|+|E|)$
- Finding if two vertices are directly connected is no longer constant time
- Which representation is best?
Graph Representation – Adjacency List

- Adjacency List: \( n \) lists, one for each vertex
- Linked list for a vertex \( u \) contains the vertices to which \( u \) has an outgoing edge
  - For directed graphs each edge appears in just one list
  - For undirected graph each edge appears in both vertex lists
- Size is \( O(|V|+|E|) \)
- Finding if two vertices are connected is no longer constant time
- Which representation is best?
  - Depends on type of graph applications
  - If dense connectivity, Matrix is usually best
  - If sparse, then List is often best
Depth-First Search of Undirected Graphs

- What parts of the graph are reachable from a given vertex
  - Reachable if there is a path between the vertices

- Note that in our representations the computer only knows which are the neighbors of a specific vertex

- Deciding reachability is like exploring a labyrinth
  - If we are not careful, we could miss paths, explore paths more than once, etc.
  - What algorithm do you use if you are at a cave entrance and want to find the cave exit on the other side?
    - Chalk or string is sufficient
    - Recursive stack will be our string, and visited(v)=true our chalk
    - Why not just "left/right-most" with no chalk/visited?
    - Depth first vs Breadth first?
### Explore Procedure

**Figure 3.3** Finding all nodes reachable from a particular node.

**Procedure** `explore(G, v)`

**Input:** \( G = (V, E) \) is a graph; \( v \in V \)

**Output:** visited\((u)\) is set to true for all nodes \( u \) reachable from \( v \)

- `visited(v) = true`
- `previsit(v)`
- for each edge \((v, u) \in E:\`
  - if not visited\((u)\): explore\((u)\)
- `postvisit(v)`

- DFS algorithm to find which nodes are reachable from an initial node \( v \)
- `previsit(v)` and `postvisit(v)` are optional updating procedures
- Complexity?
Explore Procedure

**Figure 3.3** Finding all nodes reachable from a particular node.

procedure `explore(G, v)`

Input: \( G = (V, E) \) is a graph; \( v \in V \)
Output: \( \text{visited}(u) \) is set to true for all nodes \( u \) reachable from \( v \)

\( \text{visited}(v) = \text{true} \)

\( \text{previsit}(v) \)

for each edge \( (v, u) \in E \):
  if not \( \text{visited}(u) \):
    \( \text{explore}(u) \)

\( \text{postvisit}(v) \)

- DFS algorithm to find which nodes are reachable from an initial node \( v \)
- `previsit(v)` and `postvisit(v)` are optional updating procedures
- Complexity
  - Each reachable edge \( e_r \), checked exactly once (twice if undirected)
  - \( O(|E_r| + |V|) \)
An undirected graph is *connected* if there is a path between any pair of nodes.

Otherwise graph is made up of disjoint *connected components*.

Visiting entire graph is $O(|V| + |E|)$ – Note this is the amount of time it would take just to scan through the adjacency list.

- Why not just say $O(|E|)$ since $|V|$ is usually smaller than $|E|$?
- What do we accomplish by visiting entire graph?
Previsit and Postvisit Orderings – Each pre and post visit is an ordered event

**Figure 3.3** Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
    if not visited(u): explore(u)
postvisit(v)
```

**Figure 3.5** Depth-first search.

```
procedure dfs(G)
for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v): explore(v)
```
Previsit and Postvisit Orders

- DFS yields a forest (disjoint trees) when there are disjoint components in the graph.
- Can use pre/post visit numbers to detect properties of graphs:
  - Account for all edges: Tree edges (solid) and back edges (dashed).
  - Back edges detect cycles.
- Properties still there even if explored in a different order (i.e. start with F, then J).
Depth-First Search in Directed Graphs

- Can use the same DFS algorithm as before, but only traverse edges in the prescribed direction
  - Thus, each edge just considered once (not twice like undirected)
    - still can have separate edges in both directions (e.g. (e, b))
  - Root node of DFS tree is A in this case (if we go alphabetically), Other nodes are descendants of A in the DFS tree
  - Natural definitions for ancestor, parent, child in the DFS Tree
Depth-First Search in Directed Graphs

- Tree edges and back edges (2 below) the same as before
- Added terminology for **DFS Tree** of a directed graph
  - Forward edges lead from a node to a nonchild descendant (2 below)
  - Cross edges lead to neither descendant or ancestor, lead to a node which has already had its postvisit (2 below)
**Back Edge Detection**

- Ancestor/descendant relations, as well as edge types can be read off directly from pre and post numbers of the DFS tree – just check nodes connected by each edge.
- Initial node of edge is $u$ and final node is $v$.
- Tree/forward reads $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.

<table>
<thead>
<tr>
<th>pre/post ordering for $(u, v)$</th>
<th>Edge type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$ $v$ $v$ $u$</td>
<td></td>
</tr>
<tr>
<td>$v$ $u$ $u$ $v$</td>
<td></td>
</tr>
<tr>
<td>$v$ $v$ $u$ $u$</td>
<td></td>
</tr>
</tbody>
</table>

• If G First?
DAG – Directed Acyclic Graph

- DAG – a directed graph with no cycles
- Very common in applications
  - Ordering of a set of tasks. Must finish pre-tasks before another task can be done
- How do we test if a directed graph is acyclic in linear time?
DAG – Directed Acyclic Graph

- DAG – a directed graph with no cycles
- Very common in applications
  - Ordering of a set of tasks. Must finish pre-tasks before another task can be done

How do we test if a directed graph is acyclic in linear time?

Property: A directed graph has a cycle iff DFS reveals a back edge

Just do DFS and see if there are any back edges

How do we check DFS for back edges and what is complexity?
DAG – Directed Acyclic Graph

- DAG – a directed graph with no cycles
- Very common in applications
  - Ordering of a set of tasks. Must finish pre-tasks before another task can be done
- How do we test if a directed graph is acyclic in linear time?
- Property: A directed graph has a cycle iff DFS reveals a back edge
- Just do DFS and see if there are any back edges
- How do we check DFS for back edges and what is complexity?
- \(O(|E|)\) – for edge check pre/post values of nodes
  - Could equivalently test while building the DFS tree
Every DAG can be linearized
  - More than one linearization usually possible
Every DAG can be linearized
  - More than one linearization usually possible

Algorithm to linearize
  - Property: In a DAG, every edge leads to a node with lower post number
  - Do DFS, then linearize by sorting nodes by decreasing post number
  - Which node do we start DFS at? Makes no difference, B will always have largest post (Try A and F)
  - Other linearizations may be possible

Another property: Every DAG has at least one source and at least one sink
  - Source has no input edges – If multiple sources, linearization can start with any of them
  - Sink has no output edges

Is the above directed graph connected – looks like it if it were undirected, but can any node can reach any other node?
Alternative Linearization Algorithm

Another linear time linearization algorithm

This technique will help us in our next goal

- Find a source node and put it next on linearization list
  - How do we find source nodes?
  - Do a DFS and count in-degree of each node
  - Put nodes with 0 in-degree in a source list
  - Each time a node is taken from the source list and added to the linearization, decrement the in-degree count of all nodes to which the source node had an out link. Add any adjusted node with 0 in-degree to the source list
- Delete the source node from the graph
- Repeat until the graph is empty
Strongly Connected Components

- Two nodes $u$ and $v$ in a directed graph are connected if there is a path from $u$ to $v$ and a path from $v$ to $u$.
- The disjoint subsets of a directed graph which are connected are called *strongly connected components*.

How could we make the entire graph strongly connected?
Meta-Graphs

- Every directed graph is a DAG of its strongly connected components

This meta-graph decomposition will be very useful in many applications (e.g. a high level flow of the task with subtasks abstracted)
Algorithm to Decompose a Directed Graph into its Strongly Connected Components

- `explore(G, v)` will terminate precisely when all nodes reachable from `v` have been visited.
- If we could pick a `v` from any sink meta-node then call `explore`, it would explore that complete SCC and terminate.
- We could then mark it as an SCC, remove it from the graph, and repeat.
- How do we detect if a node is in a meta-sink?
- How about starting from the node with the lowest post-visit value?
- Won't work with arbitrary graphs (with cycles) which are what we are dealing with.
Algorithm to Decompose a Directed Graph into its Strongly Connected Components

- `explore(G, v)` will terminate precisely when all nodes reachable from `v` have been visited.
- If we could pick a `v` from any sink meta-node then call `explore`, it would explore that complete SCC and terminate.
- We could then mark it as an SCC, remove it from the graph, and repeat.

How do we detect if a node is in a meta-sink?
- Property: Node with the highest post in a DFS search must lie in a source SCC.
- Just temporarily reverse the edges in the graph and do a DFS on \( G^R \) to get post ordering.
- Then node with highest post in DFS of \( G^R \), where it is in a source SCC, must be in a sink SCC in \( G \).
- Repeat until graph is empty: Pick node with highest remaining post score (from DFS on \( G^R \)), explore and mark that SCC in \( G \), and then prune that SCC.
Example

- Create $G^R$ by reversing graph $G$
- Do DFS on $G^R$
- Repeat until graph $G$ is empty:
  - Pick node with highest remaining post score (from DFS tree on $G^R$), and explore starting from that node in $G$
  - That will discover one sink SCC in $G$
  - Prune all nodes in that SCC from $G$ and $G^R$
- Complexity?
  - Create $G^R$
  - Post-ordered list – add nodes as do dfs($G^R$)
  - Do DFS with $V$ reverse ordered from PO list
    - visited flag = removed
Biconnected Components

- If deleting vertex $a$ from a component/graph $G$ splits the graph then $a$ is called a \textit{separating vertex (cut point, articulation point)} of $G$.
- A graph $G$ is \textit{biconnected} if and only if there are no separating vertices. That is, it requires deletion of at least 2 vertices to disconnect $G$.
  - Why might this be good in computer networks, etc?
  - A graph with just 2 connected nodes is also a biconnected component.
- A \textit{bridge} is an edge whose removal disconnects the graph.
- Any 2 distinct biconnected components have at most one vertex in common.
- $a$ is a separating vertex of $G$ if and only if $a$ is common to more than one of the biconnected components of $G$. 
Biconnected Algorithm – Some hints

- DFS can be used to identify the biconnected components, bridges, and separating vertices of an undirected graph in linear time.

- A is a separating vertex of G if and only if either
  1. A is the root of the DFS tree and has more than one child, or
  2. A is not the root, and there exists a child s of A such that there is no backedge between any descendent of s (including s) and a proper ancestor of A.

- \( \text{low}(u) = \min(\text{pre}(u), \text{pre}(w)) \), where (v,w) is a backedge for some descendant v of u.

- Use low to identify separating vertices, and run another DFS with an extra stack of edges to remove biconnected components one at a time.
Discovering Separating Vertices with BFS

- \( a \) is a separating vertex of \( G \) if and only if either
  1. \( a \) is the root of the DFS tree and has more than one child, or
  2. \( a \) is not the root, and there exists a child \( s \) of \( a \) such that there is no backedge between any descendent of \( s \) (including \( s \)) and a proper ancestor of \( a \).

DFS tree with pre-ordering

```plaintext
a:1
  b:2
    d:4
      e:5
  c:3
```

```
 a
  |
  v
 b
  |
  v
 c
  |
  v
 d
  |
  v
 e
```
a is a separating vertex of G if and only if either
1. \( a \) is the root of the DFS tree and has more than one child, or
2. \( a \) is not the root, and there exists a child \( s \) of \( a \) such that there is no backedge between any descendant of \( s \) (including \( s \)) and a proper ancestor of \( a \).

\[
\text{low}(u) = \min \left\{ \text{pre}(u), \text{pre}(w) \mid (v,w) \text{ is a backedge from } u \text{ or a descendant of } u \right\}
\]

DFS tree with pre-ordering and low numbering

- \( a \): 1,1
- \( b \): 2,2
- \( c \): 3,3
- \( d \): 4,3
- \( e \): 5,3

\( u \) is a sep. vertex iff there is any child \( v \) of \( u \) s.t. \( \text{low}(v) \geq \text{pre}(u) \)
a is a separating vertex of G if and only if either
1. a is the root of the DFS tree and has more than one child, or
2. a is not the root, and there exists a child s of a such that there is no backedge between any descendent of s (including s) and a proper ancestor of a.

\[
low(u) = \min\left\{ \begin{array}{l}
pre(u) \\
pre(w) \text{ where } (v,w) \text{ is a backedge from } u \text{ or a descendant of } u
\end{array} \right. 
\]

DFS tree with pre-ordering and low numbering

u is a sep. vertex iff there is any child v of u s.t.
\( low(v) \geq pre(u) \)
- $a$ is a separating vertex of $G$ if and only if either
  1. $a$ is the root of the DFS tree and has more than one child, or
  2. $a$ is not the root, and there exists a child $s$ of $a$ such that there is no backedge between any descendent of $s$ (including $s$) and a proper ancestor of $a$.

$$low(u) = \min \left\{ \begin{array}{ll}
pre(u) \\
pre(w) & \text{where } (v,w) \text{ is a backedge from } u \text{ or a descendant of } u
\end{array} \right.$$
Example with pre and low

pre numbering

a,1
b,2
g,3
c,4
e,7
d,5
f,8
h,6
Example

\[ low(u) = \min \begin{cases} 
pre(u) \\
pre(w) \text{ where } (v,w) \text{ is a backedge from } u \text{ or a descendant of } u 
\end{cases} \]

- u is a sep. vertex iff there is any child v of u s.t. low(v) \geq pre(u)
Example with pre and low

\[\text{low}(u) = \min \left\{ \frac{\text{pre}(u)}{\text{pre}(w)} \right\} \text{ where } (v,w) \text{ is a backedge from } u \text{ or a descendant of } u\]

u is a sep. vertex iff there is any child v of u s.t. \(\text{low}(v) \geq \text{pre}(u)\)