Graph basics, 1

**Definition**

An *undirected graph* consists of a set of vertices $V$ and a set of edges $E$ between vertices.


Graph basics, 2

**Definition**

An *directed graph* consists of a set of vertices $V$ and a set of (directed) edges $E$ between vertices. (So, $E \subseteq \{ (u,v) \mid u, v \in V \land u \neq v \}$.)

Note: In this course, we shall only work with *finite* graphs.

Graph basics, 3

**Adjacency Matrix Representation**

Let $V = \{1, \ldots, n\}$ and $a_{ij} = \text{true} \iff (i,j) \in E$.

Testing if $(i,j) \in E$: $O(1)$ time

Finding the vertices adjacent to $i$: $O(n)$ time
**Adjacency Matrix Representation**

Let $V = \{1, \ldots, n\}$ and $a_{ij} = \text{true} \iff (i,j) \in E$.

```
1 2 3 4
1 F T F T
2 F F F T
3 T T F F
4 F F T F
```

- Testing if $(i,j) \in E$: $O(1)$ time
- Finding the vertices adjacent to $i$: $O(n)$ time


**Adjacency List Representation**

Let $V = \{1, \ldots, n\}$ and $L_i = \text{a list of vertices adjacent to } i$.

```
1 [2, 5]
2 [1, 3, 5]
3 [2, 4]
4 [3, 5, 6]
5 [1, 2, 6]
6 [4]
```

- Testing if $(i,j) \in E$: $O(n)$ time
- Finding the vertices adjacent to $i$: $O(1)$ time

**Depth-First Exploration, 1**

```
procedure explore(G, v)
    // Input: a graph $G = (V, E)$ and $v \in V$
    // Output: for all vertices $u$, reachable from $v$: $\text{visited}[u]$ is set to true
    visited[v] ← true
    previsit(v)
    for each $u$ adjacent to $v$ do
        if not visited[u] then explore(G, u)
    postvisit(v)
```

- 3, 5, and 6 are adjacent to 4
- 3 is adjacent to 4, but neither 1 nor 2 is adjacent to 4.
**Depth-First Exploration, 2**

**Definition**

\[ u \text{ is visited } \iff \text{explore eventually sets } \text{visited}[u] \leftarrow \text{true}. \]

\[ u \text{ is unvisited } \iff \text{explore never sets } \text{visited}[u] \leftarrow \text{true}. \]

**Lemma**

Suppose initially \( \text{visited}[u] = \text{false} \) for each \( u \in V \). Then explore visits exactly all vertices reachable from \( v \).

**Proof:**

**procedure** explore(G, v)  
\[ \text{visited}[v] \leftarrow \text{true} \]  
\[ \text{previsit}(v) \]  
\[ \text{for each } u \text{ adjacent to } v \text{ do} \]  
\[ \quad \text{if not } \text{visited}[u] \text{ then explore}(G, u) \]  
\[ \text{postvisit}(v) \]

**Claim 1:** If \( u \) is not reachable from \( v \), then \( u \) is unvisited. (Why?)

**More...**

**Depth-First Exploration of the Entire Graph**

**procedure** dfs(G)  
\[ // G = (V, E) \]  
\[ \text{for each } v \in V \text{ do} \]  
\[ \quad \text{visited}[v] \leftarrow \text{false} \]  
\[ \text{for each } v \in V \text{ do} \]  
\[ \quad \text{if not } \text{visited}[v] \text{ then explore}(G, v) \]  
\[ \text{procedure} \text{ explore}(G, v) \]  
\[ \quad \text{visited}[v] \leftarrow \text{true} \]  
\[ \quad \text{previsit}(v) \]  
\[ \quad \text{for each } u \text{ adjacent to } v \text{ do} \]  
\[ \quad \quad \text{if not } \text{visited}[u] \text{ then explore}(G, u) \]  
\[ \quad \text{postvisit}(v) \]

**Run time analysis:**

- Each \( v \) is explore’d exactly once. (Why?)
- In the undirected case, each edge is explore’d down twice. (Why?)
- In the directed case, each edge is explore’d down once. (Why?)
- Under the adjacency list representation, this all takes \( \Theta(|V| + |E|) \) time. (Why?)

**Depth-First Exploration of an Undirected Graph**

**Definition**

- A **tree edge** is an edge the exploration moves down.
- A **back edge** is an edge the exploration fails to move down.
- A **DFS forest** is the forest made up of the tree edges.

**Figure from DPV**
Connected Components in an Undirected Graph

**Procedure** dfs(G)  // G = (V, E)
- for each v ∈ V do visited[v] ← false; cc[v] ← 0
- count ← 1
- for each v ∈ V do
  - if not visited[v] then explore(G, v); count ← count + 1

**Procedure** explore(G, v)
- visited[v] ← true
- previsit(v)
- for each u adjacent to v do
  - if not visited[u] then explore(G, u)
- postvisit(v)

**Procedure** previsit(v)
- cc[v] ← count

Previsit and postvisit orderings

**Procedure** previsit(v)
- pre[v] ← clock
- clock ← clock + 1

**Procedure** postvisit(v)
- post[v] ← clock
- clock ← clock + 1

**Lemma**
For any two distinct vertices u and v, either
(a) [pre[u], post[u]] ∩ [pre[v], post[v]] = ∅ or
(b) [pre[u], post[u]] ⊂ [pre[v], post[v]] = ∅ or
(c) [pre[u], post[u]] ⊃ [pre[v], post[v]] = ∅.

Depth-first search in directed graphs, 1

**Types of edges**

(a) **Tree edge**: part of the DFS forest
(b) **Forward edge**: lead to nonchild descendant in the DFS tree.
(c) **Back edge**: lead to an ancestor in the DFS tree.
(d) **Cross edge**: None of the above. They lead to a vertex that has been completely explored.

Pre/post ordering for (u, v)

- \[
\begin{array}{cccc}
  u & v & v & u \\
  \end{array}
\]
  - Tree/Forward edges
- \[
\begin{array}{cccc}
  v & u & u & v \\
  \end{array}
\]
  - Back edges
- \[
\begin{array}{cccc}
  u & u & v & v \\
  \end{array}
\]
  - Cross edges
Testing for a Cycle

Proposition

A directed graph $G$ has a cycle $\iff$ any depth-first search of $G$ finds a back edge.

- **Claim 1:** If there is a back edge, there is a cycle.  
  *Easy*
- **Claim 2:** If there is a cycle, a DFS finds a back edge.

  *Proof:*
  - Suppose $G$ has a cycle.
  - Suppose $u$ is the first vertex of this cycle a particular DFS finds.
  - Then the DFS visits all the vertices reachable from $u$.
  - In the course of this it must find a back edge.  

Topological Sorting, 1

**Definition**

(a) A dag is a directed, acyclic (i.e., no cycles) graph.
(b) Suppose $G = (V, E)$ is a dag. For each $u, v \in V$, write $u \leq_G v$ iff there is a path from $u$ to $v$ in $G$. (Note: $(u \leq_G v \& v \leq_G u) \Rightarrow u = v$.)
(c) A topological sort of a dag $G$ is ordering of $V$: $v_1, \ldots, v_n$ such that $v_i \leq_G v_j \iff i \leq j$.

Every dag has a topological sort, but how to find it?

**Proposition**

If $(u, v)$ is an edge in a dag, then \( \text{post}[u] > \text{post}[v] \).  
(Why?)

**Corollary**

Every dag has at least one source and at least one sink.  
(Why?)

source $\equiv$ no edges in sink $\equiv$ no edges out

Topological Sorting, 2

**procedure** dfs($G$) 
// $G = (V, E)$
  for each $v \in V$ do visited[$v$] $\leftarrow$ false; pre[$v$] $\leftarrow$ 0; post[$v$] $\leftarrow$ 0;
  clock $\leftarrow$ 0; topsort $\leftarrow$ emptylist
  for each $v \in V$ do
    if not visited[$v$] then explore($G$, $v$);

**procedure** explore($G$, $v$)
  visited[$v$] $\leftarrow$ true
  previsit($v$)
  for each $u$ adjacent to $v$ do
    if not visited[$u$] then explore($G$, $u$)
  postvisit($v$)

**procedure** previsit($v$)
  pre[$v$] $\leftarrow$ clock; clock $\leftarrow$ clock + 1

**procedure** postvisit($v$)
  post[$v$] $\leftarrow$ clock; clock $\leftarrow$ clock + 1; \text{add} \ v \ \text{to the front of} \ \text{topsort}

Next time: strongly connected components

Strongly Connected Components

Below $G = (V, E)$ is a directed graph.

**Definition**

We say that $u, v \in V$ are connected (written: $u \sim_G v$) $\iff$ there is a $G$-path from $u$ to $v$ and a $G$-path from $v$ to $u$.

**Lemma**

$\sim_G$ is an equivalence relation.
I.e., $u \sim_G u$ and $u \sim_G v \iff v \sim_G u$ and $(u \sim_G v \& v \sim_G w) \Rightarrow u \sim_G w$.

**Definition**

A $\sim_G$ equivalence class is called a strongly connected component of $G$.

**Definition**

$G/\sim_G = (\tilde{V}, \tilde{E})$, where $\tilde{V} = G$’s connect components and $\tilde{E} = \{ (C, C') : (\exists u \in C, v \in C')[(u, v) \in E] \}$.  

Royer (EECS)  
CIS 675 Slides  
September 24, 2009  
17 / 28
Finding Connected Components, An Example

Strongly Connected Components, An Example

A
B
C
D
E
F
G
H
I
J
K
L

Finding Connected Components, 1

Property 1
Start explore at vertex $u$.
Then explore stops after visiting exactly the vertices reachable from $u$.

Corollary
Started in a sink connected component, explore will visit exactly that component.

Q1: How to find vertex in a sink component? Q2: What to do after that?
Observation: Finding a vertex in a source component is easy.

Property 2
Do a DFS of $G$. Let $u$ be the vertex with largest $post[u]$.
Then $u$ is in the source component.

Finding Connected Components, 1

Property 1
Start explore at vertex $u$.
Then explore stops after visiting exactly the vertices reachable from $u$.

Corollary
Started in a sink connected component, explore will visit exactly that component.

Q1: How to find vertex in a sink component? Q2: What to do after that?
Observation: Finding a vertex in a source component is easy.

Property 2
Do a DFS of $G$. Let $u$ be the vertex with largest $post[u]$.
Then $u$ is in the source component.

(Why? …)
Finding Connected Components, 2

Property 2
Do a DFS of $G$. Let $u$ be the vertex with largest $\text{post}[u]$.
Then $u$ is in the source component.

Property 3
Suppose $C$ and $C'$ are SCC's and there is an edge from a vertex in $C$ to a vertex in $C'$. Then: \[
\max(\{\text{post}[v] : v \in C\}) > \max(\{\text{post}[v] : v \in C'\}).
\]

Proof Outline.
CASE: The DFS visits $C$ before $C'$.
Then the DFS visits all of $C$ and $C'$ before backing out of $C$.
CASE: The DFS visits $C'$ before $C$.
Then the DFS must visit all of $C'$ before arriving at $C$.

So we can find the source SCC, what about the sink?

Finding Connected Components, 3

Definition
$G^R = (V, \{(v, u) : (u, v) \in E\})$. \[\Rightarrow \text{in } G \Rightarrow \text{in } G^R\]

Observation: A source SSC in $G^R$ is a sink SSC in $G$.

Finding Connected Components, 4

Run time

Finding Connected Components, 5
Other Applications of DFS

- **biconnected components:**
  Suppose $G$ is undirected.
  $u \approx_G v \iff u = v$ or $u$ and $v$ are on a $G$-cycle
  The biconnected components of $G$ are the $\approx_G$-equivalence classes

- Etc. See the exercises for Chapter 3.

Other Graph Traversals

- **Breadth First Search**
  Visit $v$.
  Visit all vertices distance 1 from $v$
  Visit all vertices distance 2 from $v$
  : 
  *This is a queue-based search — DFS is stack based.*

- **Game tree search**
  The tree is too big, so you build it as you explore it.
  You have a heuristic rating function on positions
  You next explore the best-rated position not yet visited.
  *This is a priority queue based search.*

  **Next:** Paths in graphs.