The Decomposition of Graphs

DPV Chapter 3

Jim Royer

EECS

February 6, 2019
Graph basics, 1

Definition

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


For a Google-map view, click [here](http://en.wikipedia.org/wiki/Seven_bridges_of_konigsberg)—they seem to have lost a few bridges.
Definition

An directed graph consists of:

- $V$, a set of vertices and
- $E$, a set of (directed) edges between vertices.

(So, $E \subseteq \{(u,v) : u,v \in V \& u \neq v\}$.)

Note: In this course, (almost) all graphs will be finite.
Graph basics, 3

Adjacency Matrix Representation

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Diagram from http://en.wikipedia.org/wiki/Graph_(mathematics)

- 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 $a_{ij} = \text{true} \iff (i, j) \in E$.

<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>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

• 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 $L_i$ = a list of vertices adjacent to $i$.

![Graph Diagram]

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>$L_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>2</td>
<td>[1, 3, 5]</td>
</tr>
<tr>
<td>3</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 5, 6]</td>
</tr>
<tr>
<td>5</td>
<td>[1, 2, 4]</td>
</tr>
<tr>
<td>6</td>
<td>[4]</td>
</tr>
</tbody>
</table>
Adjacency List Representation
Let $V = \{1, \ldots, n\}$ and $L_i$ = a list of vertices adjacent to $i$.

- 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 $v$: $visited[u]$ is set to true

$visited[v] \leftarrow true$

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**

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

**Lemma**

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

**Proof:**

**Claim 1:** If $u$ is visited, then $u$ is reachable from $v$.

**Claim 1′:** If $u$ is not reachable from $u$, then $v$ is unvisited.

**procedure** explore$(G, v)$

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

Suppose initially visited[u] = false for each u ∈ V. Then explore visits exactly all the vertices reachable from v.

Proof (continued):

Claim 2: If u is reachable from v, then u is eventually visited.

- By way of contradiction, suppose there is an unvisited, reachable u.
- v ≠ u. (Why?)
- Take a path from v to u. [Draw the picture!]
- Let y be the last visited vertex in the path. [Draw the picture!]
- Let z be the next vertex after y on the path. [Draw the picture!]
- But by the algorithm, z must be visited, a contradiction.
- Therefore, Claim 2 and the lemma follow.
Depth-First Exploration of the Entire Graph

**procedure dfs(G)**

// G = (V, E)

for each v ∈ V do
    visited[v] ← false

for each v ∈ V do
    if not visited[v] then
        explore(G, v)

**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)
Depth-First Exploration of the Entire Graph

Graph Decomposition

Alternative graph:
Depth-First Exploration of the Entire Graph

```
procedure dfs(G)  // G = (V, E)
    for each v ∈ V do visited[v] ← false
    for each v ∈ V do
        if not visited[v] then explore(G, v)

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)
```

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 Θ(|V| + |E|) time. (Why?)
Depth-First Exploration of an Undirected Graph

Definition

(a) A tree edge is an edge the exploration moves down.
(b) A back edge is an edge the exploration fails to move down.
(c) A DFS forest is the forest made up of the tree edges.

Figures 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($v$)

$pre[v] \leftarrow \text{clock}$
$\text{clock} \leftarrow \text{clock} + 1$

**procedure** postvisit($v$)

$post[v] \leftarrow \text{clock}$
$\text{clock} \leftarrow \text{clock} + 1$

**Lemma**

For any two distinct vertices $u$ and $v$, either

(a) $[pre[u], post[u]] \cap [pre[v], post[v]] = \emptyset$ or
(b) $[pre[u], post[u]] \subset [pre[v], post[v]] = \emptyset$ or
(c) $[pre[u], post[u]] \supset [pre[v], post[v]] = \emptyset$.

Figures from DPV
Depth-first search in directed graphs, 1

DFS tree

Figure from DPV

Types of edges

- **Tree edge**: part of the DFS forest
- **Forward edge**: lead to nonchild descendant in the DFS tree.
- **Back edge**: lead to an ancestor in the DFS tree.
- **Cross edge**: None of the above. They lead to a vertex that has been completely explored.
Depth-first search in directed graphs, 2

pre/post ordering for \((u, v)\)

\[
\begin{bmatrix}
  u & v & v & u \\
  v & u & u & v \\
  u & u & v & v
\end{bmatrix}
\]

- **Tree/Forward edges**
- **Back edges**
- **Cross edges**

Figure 3.7 DFS on a directed graph.

Figure from DPV
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. *(Why?)*
Topological Sorting, 1

Definition

1. A **dag** is a directed graph that is acyclic (i.e., no cycles).
2. Suppose $G = (V, E)$ is a dag and $u, v \in V$.
   
   $u \leq_G v \iff$ there is a path from $u$ to $v$ in $G$. (★)

3. A **topological sort** of a dag $G$ is ordering of $V$: $v_1, \ldots, v_n$ such that

   \[ v_i \leq_G v_j \iff i \leq j. \]

(★) Note: $[u \leq_G v \& v \leq_G u] \Rightarrow [u = v]$. (Why?)
Topological Sorting, 2

Figure from CLRS
Definition

(a) A **dag** is a directed graph that is acyclic (i.e., no cycles).
(b) $u \leq_G v \iff \text{def there is a path from } u \text{ to } v \text{ in } G$.
(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 $\text{post}[u] > \text{post}[v]$. \hfill (Why?)

Corollary

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

- source $\equiv$ no edges in
- sink $\equiv$ no edges out
procedure dfs(G)  // G = (V, E)

clock ← 0;   topsort ← emptylist
for each v ∈ V do: visited[v] ← false; pre[v] ← 0;  post[v] ← 0
for each v ∈ V do: if not visited[v] then explore(G, v);

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)

pre[v] ← clock;  clock ← clock + 1

procedure postvisit(v)

post[v] ← clock;  clock ← clock + 1;  add v to the front of topsort
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)

pre[v] ← clock; clock ← clock + 1

procedure postvisit(v)

post[v] ← clock; clock ← clock + 1
add v to the front of topsort
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 $(u \sim_G v \land v \sim_G w) \Rightarrow u \sim_G w$.

**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 $\tilde{E} = \{(C, C') : (\exists u \in C, v \in C')[(u, v) \in E]\}$. 
Strongly Connected Components, An Example
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.
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 \( post[u] \).  
*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:** \( \max(\{ post[v] : v \in C \}) > \max(\{ post[v] : v \in C' \}) \).

**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 \quad \text{\small \( \rightarrow \) in } G \quad \Rightarrow \quad \text{\small \( \leftarrow \) in } G^R \]

**Observation:** A source SSC in \( G^R \) is a sink SSC in \( G \).

\[ \cdots \text{ We know how to find a vertex in the sink SSC of } G. \]
Finding Connected Components, 4

1. Do a DFS on $G^R$.
2. Order the vertices $v_1, \ldots, v_n$ by finishing time (biggest to smallest).
3. `count ← 1`
   `for i ← 1 to n do`
   `if not visited[v_i] then`
   `explore(G, v_i)`
   `count ← count + 1`

   `where`

   `procedure previsit(v)`
   `scc[v] ← count`

Figure from CLRS
Finding Connected Components, 5

1. Do a DFS on $G^R$.
2. Order the vertices $v_1, \ldots, v_n$ by finishing time (biggest to smallest).
3. $count \leftarrow 1$
   
   for $i \leftarrow 1$ to $n$ do
     
     if not visited[$v_i$] then
       
       explore($G, v_i$)
       
       $count \leftarrow count + 1$

where

procedure previsit($v$)

$scc[v] \leftarrow count$

---

Run time

1. $\Theta(|V| + |E|)$
2. $\Theta(|V|)$
3. $\Theta(|V| + |E|)$

∴ The total time is $\Theta(|V| + |E|)$.

(Why?)
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 (BFS)**
  
  Visit $v$.
  Visit all vertices distance 1 from $v$
  Visit all vertices distance 2 from $v$
  

  *BFS 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.*