算法设计与分析(2024年春季学期)

Graph Basics

授课老师: 栗师

南京大学计算机科学与技术系
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - \(O(n + m)\)-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s \(O(n + m)\)-Time Algorithm for Finding SCCes
Examples of Graphs

Figure: Road Networks

Figure: Social Networks

Figure: Internet

Figure: Transition Graphs
(Undirected) Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
    - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$
Abuse of Notations

- For (undirected) graphs, we often use \((i, j)\) to denote the set \(\{i, j\}\).
- We call \((i, j)\) an unordered pair; in this case \((i, j) = (j, i)\).

\[
E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}
\]
- Social Network: Undirected
- Transition Graph: Directed
- Road Network: Directed or Undirected
- Internet: Directed or Undirected
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

- **Linked lists**
  - For every vertex $v$, there is a linked list containing all **neighbours** of $v$.
  - If graph is static: store neighbors of all vertices in a length-2$m$ array, where the neighbors of any vertex are consecutive.
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ( (u, v) \in E )</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td>( O(n) )</td>
<td>( O(d_v) )</td>
</tr>
</tbody>
</table>
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Implementing BFS using a Queue

**BFS(s)**

1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as “visited” and all other vertices as “unvisited”
3: while head ≤ tail do
4:   v ← queue[head], head ← head + 1
5:   for all neighbours u of v do
6:     if u is “unvisited” then
7:       tail ← tail + 1, queue[tail] = u
8:     mark u as “visited”

- Running time: $O(n + m)$. 
Example of BFS via Queue

![Graph and Queue Diagram]

- Graph:
  - Nodes: 1, 2, 3, 4, 5, 6, 7, 8
  - Edges: 2-3, 3-5, 3-8, 7-3, 4-1, 1-2, 2-5, 2-6, 5-6

- Queue:
  - Order: 1, 2, 3, 4, 5, 7, 8, 6

- Head and Tail Indicators
  - Head:
    - Marked on the queue
  - Tail:
    - Marked on the queue

- Vertex Indicators:
  - Node 2 is marked with a red arrow pointing towards node 6.
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Implementing DFS using Recursion

**DFS(s)**
1. mark all vertices as “unvisited”
2. recursive-DFS(s)

**recursive-DFS(v)**
1. mark v as “visited”
2. for all neighbours u of v do
3. if u is unvisited then recursive-DFS(u)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
Def. A graph $G = (V, E)$ is a **bipartite graph** if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$. 
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- ...$
- Report “not a bipartite graph” if contradiction was found
- If $G$ contains multiple connected components, repeat above algorithm for each component
Test Bipartiteness

bad edges!
Testing Bipartiteness using BFS

**BFS**($s$)

1. $\text{head} \leftarrow 1$, $\text{tail} \leftarrow 1$, $\text{queue}[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. $\text{color}[s] \leftarrow 0$
4. while $\text{head} \leq \text{tail}$ do
5. \hspace{1em} $v \leftarrow \text{queue}[\text{head}]$, $\text{head} \leftarrow \text{head} + 1$
6. \hspace{1em} for all neighbours $u$ of $v$ do
7. \hspace{2em} if $u$ is “unvisited” then
8. \hspace{3em} $\text{tail} \leftarrow \text{tail} + 1$, $\text{queue}[\text{tail}] = u$
9. \hspace{3em} mark $u$ as “visited”
10. \hspace{1em} else if $\text{color}[u] = \text{color}[v]$ then
11. \hspace{2em} print(“$G$ is not bipartite”) and exit
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: \textbf{for} each vertex $v \in V$ \textbf{do}
3: \hspace{1em} \textbf{if} $v$ is “unvisited” \textbf{then}
4: \hspace{2em} test-bipartiteness($v$)
5: \hspace{1em} print(“$G$ is bipartite”)

\textbf{Obs.} Running time of algorithm = $O(n + m)$
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - \(O(n + m)\)-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s \(O(n + m)\)-Time Algorithm for Finding SCCes
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that
- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
topological-sort($G$)

1: let $d_v \leftarrow 0$ for every $v \in V$
2: for every $v \in V$ do
3: for every $u$ such that $(v, u) \in E$ do
4: $d_u \leftarrow d_u + 1$
5: $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$
6: while $S \neq \emptyset$ do
7: $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8: $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9: for every $u$ such that $(v, u) \in E$ do
10: $d_u \leftarrow d_u - 1$
11: if $d_u = 0$ then add $u$ to $S$
12: if $i < n$ then output “not a DAG”

- $S$ can be represented using a queue or a stack
- Running time $= O(n + m)$
### $S$ as a Queue or a Stack

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$\text{head} \leftarrow 1$, $\text{tail} \leftarrow 0$</td>
<td>$\text{top} \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$\text{head} \leq \text{tail}$</td>
<td>$\text{top} &gt; 0$</td>
</tr>
</tbody>
</table>
| Add($v$)      | $\text{tail} \leftarrow \text{tail} + 1$
|              | $S[\text{tail}] \leftarrow v$    | $\text{top} \leftarrow \text{top} + 1$
|              |                                   | $S[\text{top}] \leftarrow v$ |
| Retrieve $v$  | $v \leftarrow S[\text{head}]$
|              | $\text{head} \leftarrow \text{head} + 1$ | $v \leftarrow S[\text{top}]$
|              |                                   | $\text{top} \leftarrow \text{top} - 1$ |
Example

queue: \[ \begin{array}{cccccc}
  a & b & c & d & f & e & g \\
\end{array} \]

| degree | \hline
| 0      | 0      | 0      | 0      | 0      | 0      | 0      |

\[ g \]
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
**Def.** Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of $e$ from $G$ will increase its number of connected components.

- When $G$ is connected, $e \in E$ is a bridge iff its removal will disconnect $G$.

**Def.** A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting $u$ and $v$.

**Lemma** Let $B$ be the set of bridges in a graph $G = (V, E)$. Then, every connected component in $(V, E \setminus B)$ is 2-edge-connected. Every such component is called a 2-edge-connected component of $G$. 
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
Vertical and Horizontal Edges

- \( G = (V, E) \): connected graph
- \( T = (V, E_T) \): rooted spanning tree of \( G \)
- \((u, v) \in E \setminus E_T\) is
  - vertical if one of \( u \) and \( v \) is an ancestor of the other in \( T \),
  - horizontal otherwise.
$G = (V, E)$: connected graph

$T$: a DFS tree for $G$

**Q:** Can there be a horizontal edges $(u, v)$ w.r.t $T$?

**A:** No!
- $G = (V, E)$: connected graph
- $T$: a DFS tree for $G$
- $G$ contains only tree and vertical edges
- vertical edges: not bridges

Lemma
- $(u, v) \in T$, $u$ is parent
- $(u, v)$ is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of $v$ and an (inclusive) ancestor of $u$
- $v.l$: the level of vertex $v$ in DFS tree
- $T_v$: subtree rooted at $v$
- $v.r$: the smallest level that can be reached by a vertical edge from $T_v$
- $(\text{parent}(u), u)$ is a bridge if and only if $u.r \geq u.l$. 
recursive-DFS($v$)

1: mark $v$ as “visited”
2: $v.r \leftarrow \infty$
3: for all neighbours $u$ of $v$ do
4: \hspace{1em} if $u$ is unvisited then $\triangleright u$ is a child of $v$
5: \hspace{2em} $u.l \leftarrow v.l + 1$
6: \hspace{2em} recursive-DFS($u$)
7: \hspace{1em} if $u.r \geq u.l$ then claim $(v, u)$ is a bridge
8: \hspace{2em} if $u.r < v.r$ then $v.r \leftarrow u.r$
9: \hspace{2em} else if $u.l < v.l - 1$ then $\triangleright u$ is ancestor but not parent
10: \hspace{2em} if $u.l < v.r$ then $v.r \leftarrow u.l$
finding-bridges

1: mark all vertices as “unvisited”
2: for every $v \in V$ do
3:    if $v$ is unvisited then
4:       $v.l \leftarrow 0$
5:    recursive-DFS($v$)

- Running time: $O(n + m)$
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
**Def.** A vertex is a *cut vertex* of $G = (V, E)$ if its removal will increase the number of connected components of $G$.

**Def.** A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between $u$ and $v$.

**Lemma** A graph $G = (V, E)$ with $|V| \geq 3$ does not contain a cut vertex, if and only if it is biconnected.
Q: How can we find the cut vertices?

A: With a small modification to the algorithm for finding bridges.
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
- directed graph $G = (V, E)$.
- it may happen: there is a $u \rightarrow v$ path, but no $v \rightarrow u$ path.

**Def.** A directed graph $G = (V, E)$ is strongly connected if for every $u, v \in V$, there is a path from $u$ to $v$ in $G$.

**Def.** A strongly connected component (SCC) of a directed graph $G$ is a maximal strongly connected subgraph of $G$.

- Define equivalence relation: $u$ and $v$ are related if they are reachable from each other
- equivalence class $\equiv$ SCC
- After contracting each SCC, $G$ becomes a directed-acyclic (multi-)graph (DAG).
Q: How can we check if a directed graph $G = (V, E)$ is strongly-connected?

A:
- Run a traversal algorithm (either BFS or DFS) from $s$ twice, one on $G$, one on $G$ with all directions of edges reversed.
- If we reached all vertices in both algorithms, then $G$ is strongly-connected.
- Otherwise, it is not.

Q: How can we find all strongly-connected components (SCCes) of a directed graph $G$?

A: A much harder problem. Tarjan’s $O(n + m)$-time algorithm.
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n + m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. **Strong Connectivity in Directed Graphs**
   - Tarjan’s $O(n + m)$-Time Algorithm for Finding SCCes
Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree $T$,
- assuming every vertex is reachable from the root of $T$

**Q:** Can there be rightwards horizontal edges?

**A:** No!
Lemma  Suppose \( u \) and \( v \) are in the same SCC, and \( w \) is the lowest common ancestor (LCA) of \( u \) and \( v \) in \( T \). Then \( w \) is the same SCC as \( u \) and \( v \).

Proof.

Idea: using leftward, upwards and tree edges, \( u \) can not reach \( v \) without touching \( w \) or its ancestors.

Lemma  The vertices in every SCC of \( G \) induce a sub-tree in \( T \).
An Intermediate Algorithm to Keep in Mind

1. build the DFS tree $T$
2. while $T$ is not empty do
3. find the first vertex $v$ in the posterior-order-traversal of $T$
   satisfying the following property: there are no edges from $T_v$ to outside $T_v$
4. claim vertices in $T_v$ as a SCC, remove them from $T$ and all edges incident to them from $T$ and $G$
Illustration of Intermediate Algorithm
Illustration of Tarjan’s Algorithm
finding strongly connected components

1. \( \text{stack} \leftarrow \text{empty stack}, \ i \leftarrow 0 \)
2. for every \( v \in V \) do: \( v.i \leftarrow \bot, \ \text{onstack}[i] \leftarrow \text{false} \)
3. for every \( v \in V \) do
4. \quad if \( v.i = \bot \) then recursive-DFS(\( v \))

recursive-DFS(\( v \))

1. \( i \leftarrow i + 1, \ v.i \leftarrow i, \ v.r \leftarrow i \)
2. \( \text{stack}.\text{push}(v), \ \text{onstack}[v] \leftarrow \text{true} \)
3. for every outgoing edge \( (v, u) \) of \( v \) do
4. \quad if \( u.i = \bot \) then recursive-DFS(\( u \))
5. \quad if \( \text{onstack}[u] \) and \( u.r < v.r \) then \( v.r \leftarrow u.r \)
6. \quad if \( v.r = v.i \) then
7. \quad pop all vertices in \( \text{stack} \) after \( v \), including \( v \) itself
8. \quad set \( \text{onstack} \) of these vertices to be \text{false}
9. \quad declare that these vertices form an SCC
Running time of the algorithm is \( O(n + m) \).