CSE 431/531: Algorithm Analysis and Design (Spring 2022)

Graph Basics

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Properties of BFS and DFS trees
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 \( v \).
### 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 neighbors 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. Properties of BFS and DFS trees
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 $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ 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

\textbf{BFS}(s)

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

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

\[
\text{DFS}(s)
\]
1. mark all vertices as “unvisited”
2. recursive-DFS(s)

\[
\text{recursive-DFS}(v)
\]
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. Properties of BFS and DFS trees
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. head ← 1, tail ← 1, queue[1] ← $s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. color[$s$] ← 0
4. while head $\geq$ tail do
5. \hspace{1em} $v$ ← queue[tail], tail ← tail + 1
6. \hspace{1em} for all neighbours $u$ of $v$ do
7. \hspace{2em} if $u$ is “unvisited” then
8. \hspace{3em} head ← head + 1, queue[head] = $u$
9. \hspace{3em} mark $u$ as “visited”
10. \hspace{1em} else if color[$u$] = 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. Properties of BFS and DFS trees
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>$head \leftarrow 0, \ tail \leftarrow 1$</td>
<td>$top \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$head \geq tail$</td>
<td>$top &gt; 0$</td>
</tr>
</tbody>
</table>
| Add($v$)      | $head \leftarrow head + 1$
               | $top \leftarrow top + 1$
               | $S[head] \leftarrow v$
               | $S[top] \leftarrow v$
| Retrieve $v$  | $v \leftarrow S[tail]$
               | $v \leftarrow S[top]$
               | $tail \leftarrow tail + 1$
               | $top \leftarrow top - 1$ |
Example

queue: $\begin{array}{cccccccc}
  a & b & c & d & f & e & g \\
\end{array}$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

degree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

tail

head
Outline

1 Graphs

2 Connectivity and Graph Traversal
   - Testing Bipartiteness

3 Topological Ordering

4 Properties of BFS and DFS trees
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a vertical edge $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$
- Can there be a horizontal edge $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$.
- Can there be a horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$’s parent is to the right of $u$?
  - No. $v$ should be a child of $u$. 

Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$, other than the tree edges, we only have horizontal edges $(u, v)$, where

- either $u$ and $v$ are at the same level
- or $u$ is 1 level above $v$, and $v$’s parent is to the left of $u$, (or vice versa)
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
  - No.

- All non-tree edges are vertical edges.

- A vertical edge $(u, v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
**Lemma** If $G$ contains a cycle, then it has a canonical cycle.

**Proof.**
- If $G$ contains a cycle, then it must have at least one non-tree edge.
- W.r.t DFS tree $T$, we can only have vertical + tree edges
- $\exists$ at least one vertical edge
- There is a canonical cycle
- There might or might not be non-canonical ones.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
  - No.
- However, there can be horizontal edges $(u, v)$ where $u$ is visited after $v$. 
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Other than tree edges, there are two types of edges:
  - vertical edges directed to ancestors
  - horizontal edges $(u, v)$ where $u$ is visited after $v$.
- An vertical edge $(u, v)$ and the tree edges in the tree path from $v$ to $u$ form a cycle, and we call it a canonical cycle.
Properties of a DFS Tree Over a Directed Graph

**Lemma** If there is a cycle in the directed graph $G$, then there must be a canonical one.

**Proof.**
- Focus on tree edges and horizontal edges
- post-order-traversal of $T$ gives a reversed topological ordering
- Without vertical edges, $G$ has no cycles
- Again, there might be non-canonical cycles.
Algorithm 1 Check-Cycle-Directed

1: add a source $s^*$ to $G$ and edges from $s^*$ to all other vertices.
2: $visited \leftarrow$ boolean array over $V$, with $visited[v] = false, \forall v$
3: $instack \leftarrow$ boolean array over $V$, with $instack[v] = false, \forall v$
4: $DFS(s^*)$
5: return “no cycle”

Algorithm 2 $DFS(v)$

1: $visited[v] \leftarrow true, instack[v] \leftarrow true$
2: for every outgoing edge $(v, u)$ of $v$ do
3: if $inqueue[u]$ then ▷ Find a vertical edge
4: exit the whole algorithm, by returning “there is a cycle”
5: else if $visited[u] = false$ then
6: $DFS(u)$
7: $instack[v] \leftarrow false$