**CS 341: ALGORITHMS**

Lecture 10: graph algorithms I
Readings: see website

Trevor Brown  
https://student.cs.uwaterloo.ca/~cs341  
trevor.brown@uwaterloo.ca

---

**GRAPHS**

- A graph is a pair $G = (V, E)$
- $V$ contains vertices
- $E$ contains edges
  - An edge $uv$ connects two distinct vertices $u, v$
  - Also denoted $(u, v)$
- Graphs can be **undirected**
  - meaning $(u, v) = (v, u)$

**PROPERTIES OF GRAPHS**

- Number of vertices $n = |V|$
- Number of edges $m = |E| \leq n(n - 1)$
  - Note $m$ is in $\Theta(n^2)$ but not necessarily $\Omega(n^2)$
  - For undirected graphs, $m \leq \frac{n(n-1)}{2}$
  - (Asymptotically, no different)

Other common terminology:
- vertices = nodes
- edges = arcs

**DATA STRUCTURES** FOR GRAPHS

- Two main representations
  - Adjacency matrix
  - Adjacency list
- Each has pros & cons

---

**A FEW MORE TERMS**

- The **indegree** of a node $u$, denoted $\text{indeg}(u)$, is the number of edges directed into $u$
- The **outdegree**, denoted $\text{outdeg}(u)$, is the number of edges directed out from $u$
- The **neighbours** of $u$ are the nodes $u$ points to
  - Also called the **nodes adjacent to** $u$, denoted $\text{adj}(u)$

```
\text{indeg}(u) = 1
\text{outdeg}(u) = 2
\text{adj}(u) = \{1, 5\}
```
**ADJACENCY MATRIX REPRESENTATION**

- $n \times n$ matrix $A = (a_{uv})$
- rows & columns indexed by $V$
  - $a_{uv} = 1$ if $(u, v)$ is an edge
  - $a_{uv} = 0$ if $(u, v)$ is a non-edge
  - Diagonal = 0 (no self edges)

Matrix $A$

![Matrix A Diagram](image1.png)

**IMPLEMENTING AN ADJACENCY MATRIX**

Suppose we are loading a graph from input
- Assume nodes are labeled $0..n-1$
- 2D array $\text{bool} \ adj[n][n]$
- What if nodes are not labeled $0..n-1$?
  - Rename them in a preprocessing step
- What if you don’t have 2D arrays?
  - Transform 2D array index into 1D index
  - $\text{adj}[u][v] \rightarrow \text{adj}[u*n + v]$
  - (can simplify with macros in C)

**ADJACENCY LIST REPRESENTATION**

For undirected graphs
- If $\text{adj}[u]$ contains $v$ then $\text{adj}[v]$ also contains $u$

![Adjacency List Diagram](image2.png)

**IMPLEMENTING ADJACENCY LISTS**

Suppose we are loading a graph from input
- Assume nodes are labeled $0..n-1$
- Array of lists $\text{adj}[n]$
  - (In C++, something like an array of vector<int> would work)
**PROS AND CONS**

<table>
<thead>
<tr>
<th>Adjacency matrix</th>
<th>Adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to test whether ((u,v)) is an edge</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Time to list neighbours of (u)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Space complexity</td>
<td>(O(n^2))</td>
</tr>
</tbody>
</table>

- Excellent when nodes have \(O(1)\) neighbours
- Can be better for dense graphs
- Better if \(\alpha(n^2)\) edges

We call this a **sparse graph**

**COMPLEXITY**

**Naïve loop analysis:**
- \(O(n)\) iterations * \(O(\text{adj}[u])\) iterations
- \(|\text{adj}[u]| \leq n\), so \(O(n^2)\)

**BREADTH FIRST SEARCH**

A simple introduction to graph algorithms

1. **Undiscovered** nodes are white
2. **Discovered** nodes are gray
3. **Processing** adjacent edges
4. **Finished** nodes are black
5. Adjacent nodes have been processed
6. Connected graph: each node is eventually black

Example execution:

Starting at node 1

<table>
<thead>
<tr>
<th>q head</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**Complexity**

- \(O(\text{adj})\) iterations
- \(O(n)\) iterations

**Smarter loop analysis:**
- For each \(u\), iterate over all neighbours
  - We touch each edge twice (doing \(O(1)\) work each time)
  - **Total contribution** of the inner loop to the runtime: \(O(m)\)
**Smarter loop analysis:**
- Initialization time: $O(n)$
- Total contribution of the inner loop: $O(m)$
- (Over all iterations of the outer loop)
- Additional contribution of the outer loop: $O(n)$
- Total runtime: $O(m + n)$

**An analytic expression for loop complexity:**

$T_{\text{Loop}} = \sum_{u=1}^{n} \deg(u)$

$= O(n) + \sum_{u=1}^{n} \deg(u) = O(n + m)$

**Differences with adjacency matrices**
- Analysis is mostly similar
- But, it takes $O(n)$ time to determine which nodes are adjacent to $u$
- This $O(n)$ cost is paid for each $u$, resulting in a total runtime $\in O(n^2)$

---

**BFS tree**
- Connected graph: the `pred[]` array induces a tree
- The edges induced by `pred[]` are called tree edges
- Edges in the graph, but not in `pred[]`, are cross edges

**BFS: Proof of Optimal Distances**
- Denote $d_s(v)$ as the (optimal) distance between $s$ and $v$ in $G$
- Denote $d_T(v)$ as the distance between $s$ and $v$ in the BFS tree $T$
- Recall: $dist[v]$ is a value set by BFS for each node $v$

**Distance in Graph $G$ and BFS Tree $T$**

![BFS Tree Diagram](image)

**Proof Idea**

Plan: prove this in two parts

1. $dist[v] = d_s(v)$
2. $d_T(v) = d_s(v)$

Want to show: at the end of BFS, $dist[v] = d_s(v)$ for all $v$
**SKETCH OF CLAIM 1**: $\text{dist}[v] = d_T(v), \forall v \in V$

1. **Key observation**: whenever we set $\text{dist}[u] = \text{dist}[v] + 1$, $u$ is the parent of $v$ in the BFS tree.

   Based on this observation, a simple inductive proof shows $\text{dist}[v] = d_T(v)$.

2. **Claim**: $\forall v \in V$.

**SKETCH OF CLAIM 2**: $d_T(v) = d_G(v)$

- **Part 1**: $\forall v, d_G(v) \geq d_T(v)$
- **Part 2**: $\forall v, d_G(v) \geq d_T(v)$

**APPLICATION**: FINDING SHORTEST PATHS
HOW TO OUTPUT AN ACTUAL PATH
- Suppose you want to output a path from \( s \) to \( v \) with minimum distance (not just the distance to \( v \))
- Algorithm (what do you think?)
  - Similar to extracting an answer from a DP array!
  - Work backwards through the predecessors
  - Note: this will print the path in reverse! Solution?

**APPLICATION:**

**UNDIRECTED CONNECTED COMPONENTS**

**CONNECTED COMPONENTS**

Example: undirected graph with three components

Can you think of a way to use BFS to count how many connected components there are?
CONNECTED COMPONENTS

BreadthFirstSearch(V, adj, i)
BreadthFirstSearch(V, adj, j)
BreadthFirstSearch(V, adj, k)

Can be done in $O(n + m)$ time.
Complexity?

Modified BFS that (1) reuses the same colour array for consecutive calls and (2) sets comp[i] = compNum for each node i it visits.

BONUS SLIDES

ANSWER TO BFS TREE PROPERTY EXERCISE...

Defined = back edge