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 or directed
• meaning \((u, v) \neq (v, u)\)

PROPERTIES OF GRAPHS
• Number of vertices \( n = |V| \)
• Number of edges \( m = |E| \leq \binom{n}{2} \)
• 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]

ADJACENCY LIST REPRESENTATION

- For undirected graphs
- $a_{uv} = 1$ if $(u, v)$ or $(v, u)$ is an edge
- Matrix is symmetric $A^T = A$

![Adjacency List]

IMPLEMENTING AN ADJACENCY MATRIX

- Suppose we are loading a graph from input
  - Assume nodes are labeled 0..n-1
  - 2D array 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
    - adj[u][v] $\rightarrow$ adj[u*n + v]
      (can simplify with macros in C)

![Adjacency Matrix Implementation]

ADJACENCY LIST REPRESENTATION

- $n$ linked lists, one for each node
  - We write adj[u] to denote the list for node $u$
  - adj[u] contains the labels of nodes it has edges to

![Adjacency List Implementation]

IMPLEMENTING ADJACENCY LISTS

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

<table>
<thead>
<tr>
<th></th>
<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>
<td>(O(\text{outdeg}(u)))</td>
</tr>
<tr>
<td>Time to list neighbours of (u)</td>
<td>(O(n))</td>
<td>(O(\text{outdeg}(u)))</td>
</tr>
<tr>
<td>Space complexity</td>
<td>(O(n^2))</td>
<td>(O(n + m))</td>
</tr>
</tbody>
</table>

Better if \(\theta(n^2)\) edges

We call this a sparse graph

Excellent when nodes have \(\theta(1)\) neighbours

Can be better for dense graphs

BREADTH FIRST SEARCH

A simple introduction to graph algorithms

**Example execution**

Starting at node 1

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**COMPLEXITY**

\(O(n)\) (with adjacency lists)

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

- Smart iteration 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: \(O(n^2)\)
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: $O(m + n)$

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 $\text{pred}[\cdot]$ array induces a tree
• The edges induced by $\text{pred}[\cdot]$ are called tree edges
• Edges in the graph, but not in $\text{pred}[\cdot]$, are cross edges

Connected graph: the $\text{pred}[\cdot]$ array induces a tree

Graph
Tree edge
Cross edge (not part of tree)

Distance in graph $G$ and BFS tree $T$
• Denote $d_G(v)$ 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 $\text{dist}[v]$ is a value set by BFS for each node $v$

Proof idea
Want to show: at the end of BFS, $\text{dist}[v] = d_G(v)$ for all $v$

Plan: prove this in two parts
Claim 1: $\text{dist}[v] = d_G(v)$
Claim 2: $d_T(v) = d_G(v)$
**SKETCH OF CLAIM 1**: \( \text{dist}[v] = d_T(v), \forall v \in V \)

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

Based on this observation, a simple inductive proof shows \( \text{dist}(v) = d_T(v) \) (for example, by strong induction on the nodes in the order their \( \text{dist} \) values are set --- left as an exercise).

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

- **Part 1**: \( \forall v, d_T(v) \leq d_G(v) \)
  - There is a unique path \( v \to \cdots \to s \) in \( T \)
  - And \( T \) is a subgraph of \( G \)
  - So that same path also exists in \( G \) (technically reversed)

To prove \( = \) we show \( \leq \) and \( \geq \):

- **Part 2**: \( \forall v, d_G(v) \geq d_T(v) \)
  - Partition \( T \) into levels \( V_i = \{ v : d_T(v) = i \} \) by distance from \( s \)
  - Claim: there is no "forward" edge in \( G \) that "skips" a level from \( V_i \) to \( V_j, j \geq i + 2 \)
  - Suppose there is, for contradiction...

**BFS TREE PROPERTIES**

Fact: there are no "back" edges in undirected graphs that "skip" a level going up in the BFS tree.

**APPLICATION:**

**FINDING SHORTEST PATHS**

Exercise: what about directed graphs?

Answer: in both cases...
**User interfaces:**
rubber banding a mouse cursor around obstacles.

**Game AI:**
path finding in a grid graph.

**How to represent a grid graph?**
Starting to get into the details.

**BFS from here**

**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

**How to represent a grid graph?**
Dance At: push finding in a grid graph.

**Score:** 0
CONNECTED COMPONENTS

BreadthFirstSearch(V, adj, 1)
BreadthFirstSearch(V, adj, 3)
BreadthFirstSearch(V, adj, 4)

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

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

BONUS SLIDES

Answer to BFS tree property exercise...

Dotted = back edge