**CS 341: ALGORITHMS**

Lecture 12: graph algorithms

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 \( (u, v) \neq (v, u) \)

**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 \( \text{adj}(u) \)

\[
\begin{align*}
\text{indeg}(u) &= 1 \\
\text{outdeg}(u) &= 2 \\
\text{adj}(u) &= \{1, 3\}
\end{align*}
\]

**DATA STRUCTURES FOR GRAPHS**

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

**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**

---

**Directed vs Undirected Graphs**

- Directed graph
- Undirected graph

---

**2022-06-03**
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 MATRIX REPRESENTATION

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

head

tail

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 LIST REPRESENTATION

- For undirected graphs
- If `adj[u]` contains $v$ then `adj[v]` also contains $u$

head

tail

ADJACENCY LIST REPRESENTATION

- 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>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>

Adjacency matrix is useful when nodes have \(O(1)\) neighbours. Adjacency list is better for dense graphs.

We call this a sparse graph.

**TIME TO TEST WHETHER \((u,v)\) IS AN EDGE**

- \(O(1)\)
- \(O(\text{outdeg}(u))\)

**TIME TO LIST NEIGHBOURS OF \(u\)**

- \(O(n)\)
- \(O(\text{outdeg}(u))\)

**SPACE COMPLEXITY**

- \(O(n^2)\)
- \(O(n+m)\)

Better if \(\omega(n^2)\) edges.

We call this a sparse graph.

**EXCELLENT WHEN NODES HAVE \(O(1)\) NEIGHBOURS**

**BREADTH FIRST SEARCH**

A simple introduction to graph algorithms

**COMPLEXITY**

- \(O(n)\) (with adjacency list)
- **Naive loop analysis:**
  - \(O(n)\) iterations
  - \(O(|\text{adj}(u)|)\) iterations
  - \(|\text{adj}(u)| \leq n\), so \(O(n^2)\)

**SMARTER LOOP ANALYSIS:**

- For each \(u\), iterate over all neighbours
  - \(O(\text{outdeg}(u))\)
  - Total contribution of the inner loop to the runtime \(O(n)\)

**EXAMPLE EXECUTION**

Starting at node 1

- \(q:\) 1
- \(q\) head
- \(q\) tail
- \(q\) queue
- \(q\) dequeue
- \(q\) enqueue
- \(q\) grey
- \(q\) white
- \(q\) processed
- \(q\) finished
- \(q\) black

**ASSUMING ADJACENCY LIST REPRESENTATION**

- Undiscovered nodes are white
- Discovered nodes are grey
- Processing adjacent edges
- Finished nodes are black
- Adjacent nodes have been processed
- Connected graph: each node is eventually black

**START PROCESSING NODE \(u\)’S EDGES**

**DISCOVER (ENQUEUE) NEIGHBOUR \(v\)**

**FINISH PROCESSING \(u\)**

- \(O(\text{outdeg}(u))\)
- \(O(1)\)
- \(O(n)\) iterations
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:
$$T_{LOOP} \in O(\sum_{u=1}^{n} deg(u)) = O(n + m),$$
$$deg(u) = O(n) + \sum_{u=1}^{n} deg(u) = O(n^2)$$

**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 of $O(n^2)$

**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

**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: $dist[v]$ is a value set by BFS for each node $v$

**PROOF IDEA**
- Want to show: at the end of BFS, $dist[v] = d_G(v)$ for all $v$
  - Plan: prove this in two parts
    - Claim 1: $dist[v] = d_T(v)$
    - Claim 2: $d_T(v) = d_G(v)$
CLAIM 1

**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 $\text{dist}_v = \text{dist}_T(v)$ (for example, by strong induction on the nodes in the order their dist values are set—left as an exercise).

CLAIM 2

- $\text{dist}_T(v) \leq \text{dist}_G(v)$
- $\text{dist}_T(v) \geq \text{dist}_G(v)$

**Part 1:** $\forall v, \text{dist}_v \leq \text{dist}_T(v)$
- There is a unique path $v \rightarrow \cdots \rightarrow s$ in $T$.
- And $T$ is a subgraph of $G$.
- So that same path also exists in $G$ (technically reversed).

**Part 2:** $\forall v, \text{dist}_v \geq \text{dist}_T(v)$
- We've just argued that there is no "forward" edge in $G$ that "skips" a level in $T$ from $V_i$ to $V_j$, $j \geq i + 2$.
- Since no edge in $G$ "skips" a level in $T$, we know at least one edge in $G$ is needed to traverse each level between $s \in V_0$ and $p \in V_{\text{dist}_T(p)}$.
- There are $d_{\text{dist}_G}$ such levels, so $\text{dist}_G \geq \text{dist}_T(v)$.

**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
**User interfaces:** rubber banding a mouse cursor around obstacles

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

--

**How to represent a grid graph?**

--

**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?
BreadthFirstSearch(V, adj, 1)
BreadthFirstSearch(V, adj, 2)
BreadthFirstSearch(V, adj, 3)
BreadthFirstSearch(V, adj, 4)

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

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

\[
\begin{aligned}
\text{UndirectedConnectedComponents}: &\text{adj} &\text{adj} \\
\text{color}[\cdot] &\leftarrow \text{white} &\text{white} \\
\text{comp}[\cdot] &\leftarrow \{0, \ldots, 0\} \\
\text{compNum} &\leftarrow 1 \\
\text{for} &\text{start} = 1 &\text{while} \\
\text{if} &\text{color[start] is white} &\text{BFS(adj, start, colour, comp, compNum)} \\
\text{return} &\text{comp} \\
\end{aligned}
\]

BONUS SLIDES

ANSWER TO BFS TREE PROPERTY EXERCISE...