CS 341: ALGORITHMS

Lecture 12: graph algorithms I

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
GRAPHS
**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 \( m = |E| \leq n(n - 1) \)
  
  • Note \( m \) is in \( O(n^2) \) 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
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 $\text{adj}(u)$

\[
\begin{align*}
\text{indeg}(u) &= 1 \\
\text{outdeg}(u) &= 2 \\
\text{adj}(u) &= \{1,5\}
\end{align*}
\]
DATA STRUCTURES FOR GRAPHS

• Two main representations
  • Adjacency matrix
  • Adjacency list
• Each has pros & cons
**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 $A$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Graph Diagram](image)
**ADJACENCY MATRIX REPRESENTATION**

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

![Graph and adjacency matrix](image-url)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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] → adj[u*n + v]
    (can simplify with macros in C)
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 REPRESENTATION

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

Excellent when nodes have \(O(1)\) **neighbours**

Can be better for dense graphs

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

We call this a **sparse** graph

13
BREADTH FIRST SEARCH
A simple introduction to graph algorithms
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)

    colour[u] = black

return colour, pred, dist

Assuming adjacency list representation

- Undiscovered nodes are white
- Discovered nodes are gray
  - Processing adjacent edges
    - Adjacent nodes have been processed
- Finished nodes are black
- Connected graph: each node is eventually black
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)

    colour[u] = black

    return colour, pred, dist
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue
    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)
    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
    colour[u] = black
    return colour, pred, dist

COMPLEXITY

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

\(O(n)\) (with adjacency lists)
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue
    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)
    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black
    return colour, pred, dist

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: \( O(m) \)
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
dist[s] = 0
q.enqueue(s)

while q is not empty
    u = q.dequeue()
    for v in adj[u]
        if colour[v] = white
            pred[v] = u
            colour[v] = gray
dist[v] = dist[u] + 1
q.enqueue(v)
colour[u] = black

return colour, pred, dist

• 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:

$$T_{LOOP}(n) \in O\left(\sum_{u=1}^{n} (1 + \deg(u))\right)$$

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

Careful: we will also see DFS trees, and cross edges will be defined differently
BFS: PROOF OF OPTIMAL DISTANCES
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 $v$

$G$

$T$

$G$

$T$

$\begin{align*}
\text{dist}[s] &= 0 \\
\text{dist}[3] &= 1 \\
\text{dist}[5] &= 2 \\
\text{dist}[v] &= 3 \\
\end{align*}$

$\begin{align*}
d_G(v) &= 3 \\
d_T(v) &= 3 \\
\end{align*}$
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: \( \text{dist}[v] = d_T(v) \)
Claim 2: \( d_T(v) = d_G(v) \)
**SKETCH OF CLAIM 1: dist[v] = d_T(v), ∀v ∈ V**

```
BreadthFirstSearch(V[1..n], adj[1..n], s)
  pred[1..n] = [null, null, ..., null]
  dist[1..n] = [infty, infty, ..., infty]
  colour[1..n] = [white, white, ..., white]
  q = new queue

colour[s] = gray
dist[s] = 0
q.enqueue(s)

while q is not empty
  u = q.dequeue()
  for v in adj[u]
    if colour[v] = white
      pred[v] = u
      colour[v] = gray
      dist[v] = dist[u] + 1
      q.enqueue(v)
  colour[u] = black

return colour, pred, dist
```

**Key observation:** whenever we set \(dist[v] \leftarrow dist[u] + 1\), \(u\) is the parent of \(v\) in the BFS tree.

Based on this observation, a simple inductive proof shows \(dist[v] = d_T(v)\) (for example, by strong induction on the nodes in the order their \(dist\) values are set---left as an exercise).
SKETCH OF CLAIM 2: $d_T(v) = d_G(v)$

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

To prove $=$ we show $\leq$ and $\geq$
**SKETCH OF CLAIM 2:** \( d_T(v) = d_G(v) \)

- **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...

What are the consequences of "skipping" a level in \( T \)?

That "skip" edge in \( T \) looks like this in \( G \)

But that edge in \( G \) would cause 7 to have \( s \) as its parent, so \( dist[7] \) would be **only 1 greater** than its parent...

Contradicts(!) the assumption that the edge points to a node with greater distance by at least 2

**Diagram:**

- \( s \) is the root node.
- \( V_0 \) is in level 0.
- \( V_1, V_2, V_3 \) are in levels 1, 2, and 3 respectively.
- Edges are labeled with node names.
SKETCH OF **CLAIM 2**: $d_T(v) = d_G(v)$

- **Part 2**: $\forall v, d_G(v) \geq d_T(v)$
  - 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 $v \in V_{d_T(v)}$
  - There are $d_T(v)$ such levels, so $d_G(v) \geq d_T(v)$
Fact: there are no “back” edges in undirected graphs that “skip” a level going up in the BFS tree.

Exercise: what about directed graphs?

Answer in bonus slides...
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?

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?
Each time you visit a predecessor, push it into a stack.

I.e., push $v = 5$, then push $\text{pred}[v] = 4$, then push $\text{pred}[\text{pred}[v]] = 3$, then 2, ...

At the end, \textbf{pop} all off the stack. This gives 0, 1, 2, ..., 5 = \textbf{the path}!
APPLICATION:
UNDIRECTED CONNECTED 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, 3)
BreadthFirstSearch(V, adj, 4)

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

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

Complexity?

```python
UndirectedConnectedComponents(adj[1..n])
colour[1..n] = [white, ..., white]
comp[1..n] = [0, ..., 0]
compNum = 1
for start = 1..n
    if colour[start] is white
        BFS(adj, start, colour, comp, compNum)
return comp
```
BONUS SLIDES
ANSWER TO BFS TREE PROPERTY EXERCISE...