Graph Algorithms

Graphs and their representation:

- A graph $G = (V, E)$ consists of a set of vertices $V = \{v_i \mid i = 1, 2, \cdots, n\}$ and a set of edges $E = \{e_j \mid j = 1, 2, \cdots, m\}$ that connect the vertices.
  - An edge $e$ may be represented by the pair $(u, v)$ where $u$ and $v$ are the vertices being connected by $e$.
    - Depending on the problem, the pair may be ordered or unordered.

- Many problems can be represented as graphs:
  - Traveling Salesman Problems
  - Airline flights
  - Friends who know each other
  - Moves in a game (each node = the state of the board or game).
Directed and Undirected Graphs

- An undirected graph, $G = (V, E)$ is a pair:
  - $V = \text{set of distinct vertices.}$
  - $E = \text{set of edges; each member is a set of 2 vertices}$
    - For example:
      $$V = \{t, u, v, w, x, y, z\}$$
      $$E = \{\{u, v\}, \{u, w\}, \{v, w\}, \{v, y\}, \{x, z\}\}$$

- A directed graph, $G = (V, E)$:
  - Is the same as an undirected graph except $E$ is a set of ordered pair:
    - For example:
      $$E = \{(a, b), (a, c), (c, b), (b, e), (e, b)\}$$
      (1)
More Definitions

- **Weighted graphs:**
  - A weighted graph has value assigned to each of its edges.
  - More formally, there is a weight function \( w : E \rightarrow R \).
    - \( R = \) real numbers.
    - Depending on the syntax being used, we may see this as \( w(e) \) with \( e \in E \) or \( w(u, v) \) with \( u \) and \( v \) representing the vertices for the edge.

- **Degree:**
  - The degree of vertex \( v \), denoted by \( \text{deg}(v) \), is the number of edges that meet at \( v \).
  - Let \( v \) be a vertex in a directed graph \( G \). The number of vertices of \( G \) adjacent **from** \( v \) is called the outdegree of \( v \). the number of vertices of \( G \) adjacent **to** \( v \) is called the indegree of \( v \).
Representations of Graphs

- **Adjacency matrix:**
  - $M[i,j] = 1$ if $i$ and $j$ are neighbours, 0 otherwise.
  - Assign each vertex an integer index (in this example, $t = 7$, $u = 2$, etc.)
  - Assumes that any other info for a vertex and/or edge is in another data structure.
  - The matrix is symmetric for an undirected graph.
  - For a directed graph we will likely have an asymmetric matrix.

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Representations of Graphs

- **Adjacency matrix notes:**
  - Blank row: no neighbours i.e. isolated vertex.
  - $M[i, i] = $ self-loop.
  - Undirected graphs are symmetrical

- **Space**
  - $|V|^2$ bits
  - $(|V|^2 + |V|)/2$ (if undirected, but harder to actually implement).
  - Additional information, such as cost of an edge, could be stored in the matrix. Another option is to store a pointer to this information.

- **Cost of operations**
  - Are vertices $i$ and $j$ adjacent? $O(1)$
  - Add or delete edge: $O(1)$
  - Add vertex: increases size of matrix.
  - Find neighbours of $v$: $O(|V|)$. 

E. Zima (SCS, UW)
Module 7: Graph Algorithms
Winter 2020
Representations of Graphs

Undirected Example

- u → v → w
- v → u → w → y
- w → u → v
- x → z
- y → v
- z → x
- t

Directed

- a → b → c
- b → e
- c → b
- d
- e → b
- f → b

(2)
Representations of Graphs

- Adjacency list notes:
  - Space:
    - $O(|V| + |E|)$
    - Usually much smaller than $O(|V|^2)$ for a sparse graph.

- Cost of operations:
  - Add an edge $O(1)$
  - Delete an edge: Search lists for each endpoint $O(|V|)$.
  - Add vertex: Depends on the implementation.
  - Find neighbours $O($number of neighbours$)$ (better than Adj. Matrix)
  - Are $i, j$ adjacent? Search the list (worse than Adj. Matrix)
More Notation

- **Path:**
  - Given the graph \( G(V, E) \), a path from vertex \( u \) to vertex \( v \) is a sequence of vertices \((v_0, v_1, \cdots, v_k)\) such that \( v = v_0 \), \( u = v_k \), and \((v_{i-1}, v_i)\) is in \( E \) for all \( i = 1, 2, \cdots, k \).

- **Simple path:**
  - A path is simple if all the vertices in the path are distinct.

- **Cycle:**
  - A path \((v_0, v_1, \cdots, v_k)\) forms a cycle if \( v_0 = v_k \), and the path contains at least one edge.
  - A graph with no cycles is acyclic.

- **Connected graphs:**
  - An undirected graph is connected if every pair of vertices is connected by a path.
BFS and DFS

- Breadth-first, depth-first search
- Each starts at an arbitrary node, explores the whole connected component
- Assume whole graph is connected
- General view: vertices start out coloured white (not visited)
- A visited vertex is coloured gray (visited, but may have white neighbours)
- When all neighbours of a vertex are visited, it is coloured black.
General view of searching

- Gray nodes form a “frontier”
- Can choose any neighbour of a gray node to be next visited
- In general, want to perform computation
  - Preprocess when colouring gray
  - Postprocess when colouring black
  - Analogous to tree traversal uses
Breadth-first search

- Use queue (first-in, first-out) to store gray nodes
- Start by taking any vertex, colouring it gray, add it to queue
- Repeat: find white node adjacent to head of queue, colour it gray and add it to queue
- When you can’t find any such node, remove the head of the queue and colour it black
Pseudocode for BFS

colour_all_vertices_white()
while there is a white vertex s do
    BFS_tree(s)

BFS_tree(s: vertex)
colour s gray (visited)
enqueue(Q,s)
while Q not empty do
    u <- dequeue(Q)
    for each v adjacent to u
        if v white then
            colour v gray
            enqueue(Q,v)
            \((u,v)\) is a tree edge
        else
            \((u,v)\) is non-tree edge
            colour u black
Analysis of BFS

- Assume adjacency list representation
- Each vertex enqueued once (colour changes from white to gray) and dequeued once (colour changes from gray to black)
- Body of inner loop takes $\Theta(1)$ time
- Inner loop implemented by scanning adjacency list of head of queue
- Therefore each edge looked at exactly twice
- Running time is $\Theta(|V| + |E|)$ or $\Theta(n + m)$
BFS trees

- BFS finds a spanning tree of the graph (edges representing first visits)
- Nontree edges are called cross edges
- A cross edge cannot connect a node to its ancestor in the tree
- A cross edge cannot connect a node to its descendant in the tree
- These are useful in proving properties of BFS searches
Single-source shortest path

- Can use BFS to compute distances $\delta(s, v)$ from source $s$ to all other vertices $v$
- Compute quantity $d[v]$ in following way
  - $d[s] \leftarrow 0$
  - When adding $v$ to queue because $u$ is at head and there is an edge $(u, v)$, set $d[v] \leftarrow d[u] + 1$
Pseudocode for shortest path

```
colour_all_vertices_white()

while there is a white vertex s do
    BFS_tree(s)

BFS_tree(s: vertex)
    colour s gray (visited)
    enqueue(Q,s); d[s] <-- 0
    while Q not empty do
        u <- dequeue(Q)
        for each v adjacent to u
            if v white then
                colour v gray
                enqueue(Q,v); d[v] <-- d[u]+1
                \( (u,v) \) is a tree edge
            else
                \( (u,v) \) is non-tree edge
                colour u black
```
Why does this work?

**Lemma 1**
For any edge \((u, v)\), we have \(\delta(s, v) \leq \delta(s, u) + 1\)

**Lemma 2**
For all \(v\), \(d[v] \geq \delta(s, v)\).

Proof: By induction on number of enqueues
At beginning, \(d[s] = 0 = \delta(s, s)\) Suppose \(u\) is being visited and adjacent white \(v\) discovered.

\[
\delta(s, v) \leq \delta(s, u) + 1 \leq d[u] + 1 = d[v].
\]
Why does this work?

Lemma 2 proves $d[v] \geq \delta(s, v)$.

**Lemma 3**

At any point, the $d$-values of vertices in the queue are either $i$ or $i + 1$ for some $i$, and all the $i$ values are in front of all the $i + 1$ values.

\[ \begin{array}{cccccc}
  i+1 & \ldots & i+1 & i & \ldots & i & i & i \\
  \text{front} & & & & & & & \\
  & & & & & & & u
\end{array} \]
Why does this work?

Proof: By induction on number of queue operations
True at beginning (only one item in queue)
True after $u \leftarrow \text{dequeue}(Q)$
True after $\text{enqueue}(Q, v)$, because $d[v] = d[u] + 1$.

Corollary 4

$d$ values are assigned in increasing order.
Correctness continued

**Theorem 5**

\( d[v] = \delta(s, v) \)

Let \( v \) be closest vertex to \( s \) with wrong \( d \);
\( d[v] > \delta(s, v) \)

Let \( u \) be vertex just before \( v \) on shortest \( s - v \) path
\( \delta(s, v) = \delta(s, u) + 1, d[u] = \delta(s, u) \), so \( d[v] > d[u] + 1 \).

What colour is \( v \) when \( u \) is dequeued?

- White? Then it should have been visited from \( u \)
- Black? Then \( u \) should have been visited from \( v \)
- Gray? Then it was visited from some \( w \), \( d[v] = d[w] + 1 \), and \( d[w] \leq d[u] \), which implies \( d[v] \leq d[u] + 1 \) contradicting above inequality.

Thus no such \( v \) exists.
BFS application

- Connected components via BFS ...
- Presence of cycles ...
- Bi-partite graphs ...
- ...

E. Zima (SCS, UW)