Graph Algorithms

A directed graph (digraph) is a pair \( G = (V, E) \) where

\( V \) is a set of vertices

\( V = \{v_1, \ldots, v_n\} \) (n vertices) and

\( E \subseteq V \times V \) is the set of edges

\( (u, v) \in E \)

\( v \) is adjacent to \( u \).

\( v \) is adjacent from \( u \).

2 reps as a data struct:

Adjacency Matrix:

\[
A[1 \ldots n, 1 \ldots n] = \begin{cases} 
1 & \text{if } (v_i, v_j) \in E \\
0 & \text{otherwise}
\end{cases}
\]
only deal with **simple**
graphs (no self-loops,
i.e., \( A[i,i] = 0 \) for
\( 1 \leq i \leq n \)) and no
multiple edges between
vertices.

**Adjacency List (Edgelist)**

Array \( V[1..n] \)

\( V[i] \) is head of a
linked list of vertices
adjacent from \( V_i \)

adj Matrix

adj list rep is usually preferred.

\( n \) vertices
\( m \) edges

**size of adj Matrix:**
\( \Theta(n^2) \)

**""" list:** \( \Theta(n+m) \)

each edge corresponds to a unique list
item
augmentable with other info, edge weights, vertex weights, etc.

undirected graphs:

\[ u \leftrightarrow_v \text{edge} \]

adj relation is symmetric, represent as digraph as follows:

\[ u \rightarrow_v \]

\[ G^T : \text{transpose of digraph } G \]

\[ u \rightarrow_v \iff u \leftarrow_v \text{ in } G^T \]

adj matrix for \( G^T \) is the transpose of the adj matrix for \( G \).

Exercise: Given adj list rep for \( G \), produce an adj list rep for \( G^T \) in time \( \Theta(n+m) \).

Graph Search

Given initial vertex \( V \) systematically follow edges to find other vertices reachable from \( V \) and we visit them.
Box $B$ is some data structure that holds the gray vertices.

$B$ must support:

- $\text{Make}(B)$: initialize $B$ empty
- $\text{Empty}(B)$: test if $B$ is empty
- $\text{Insert}(x, B)$: insert item $x$ into $B$
- $\text{Delete}(B)$: remove and return an item from $B$

**GenericSearchAt**($G, v$)

// searches all vertices
// reachable from $v$
// assumes $\text{color}[v] = \text{white}$
// $B$ is a local box

$\text{Make}(B)$

$\text{Start}(v; \text{nil})$

Start called on $v$ when color changes from white to gray
\text{color}[v] := \text{gray} \\
\text{Insert}(v, B) \\
\text{Update}(v; \text{nil}, B) \\
\text{called whenever we encounter a gray vertex}

\text{repeat} \\
\quad u := \text{Delete}(B) \\
\text{Finish}(u) \\
\text{last processing of a vertex}
\text{color}[u] := \text{black} \\
\text{for each } w \text{ adjacent from } u \text{ do} \\
\quad \text{// traverse edge } (u, w) \\
\quad \text{if } \text{color}[w] = \text{white then} \\
\quad \quad \text{// u "discovers" w} \\
\quad \quad \text{Start}(w; u) \\
\quad \quad \text{color}[w] := \text{gray} \\
\quad \quad \text{Insert}(w, B) \\
\quad \text{if } \text{color}[w] = \text{gray then} \\
\quad \quad \text{Update}(w; u, B) \\
\text{until Empty}(B)
Start \( |V| \)
Update \( |E| + 1 \)
Finish \( |V| \)

Breadth-First Search (BFS):

- \( B \) is a simple queue (FIFO)
- process \( V \)
  - then neighbors of \( V \)
    - then neighbors of neighbors, etc.
- vertices are finished in increasing order of distance (unweighted) from \( V \)

Use BFS to find unweighted distance from \( V \):

- \( B \) queue
- Update, Finish are no-ops.

\[
\text{Start}(w \cdot u) \\
\text{if } u = \text{nil} \text{ then} \\
\quad d[w] := 0 \\
\text{else } d[w] := d[u] + 1
\]
Depth-First Search (DFS)

B is a stack (LIFO)