**Graph Algorithms:**

A graph $G$ is $(V, E)$

- $V$ = set of vertices
- $E$ = set of edges

Example: Un-directed

$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, ad, cd, be, de\}$

Adjacency:

- $\text{Adj}(a) = \{b, c\}$
- $\text{Adj}(c) = \{a, d\}$

$|V| = n$

$|E| = m$

Directed:

$u \rightarrow v$ = undirected.

$u \leftarrow v$ = undirected.

Example: Directed

$V = \{a, b, c, d, e\}$

$E = \{(a, b), (c, a), (b, d), (c, d), (d, e), (e, b)\}$

Applications: Social network, road network, interhyperlink graph

Observations:

$(n-1) \leq m \leq O(n^2)$

- If $G$ is connected, then:
  - A tree (no cycle)
  - A cycle (cm-1) edges

Basic concepts: path, connected, cycle, ---

Representation:

1. Adjacency Matrix: An $n \times n$ matrix

- If $G$ undirected
- If $G$ directed

$A[i, j] = 1$ if $(i, j) \in E$

$= 0$ (dense graphs)
(it is undirected
\( A \) is symmetric)
\( \forall u, v \) \hspace{1cm} = 0 \hspace{1cm} (\text{dense graphs})

\( O(n^2) \) space, lookups are \( O(1) \) of edges.

2. Adjacency list:
For each node \( u \in V \), \( \text{Adj}(u) = \{ v \mid (u,v) \in E \} \)

\[
\begin{array}{c}
    a & \rightarrow b \\
    b & \rightarrow d \\
    c & \rightarrow a, d \\
    d & \rightarrow b \\
    e & \\
\end{array}
\]

space: \( O(n + \sum_{u \in V} |\text{Adj}(u)|) \)
\[
= O(n + \sum_{u \in V} \deg(u))
\]

(look-up time
\( \text{could be } O(1) \))
\( = O(n+m) \) \hspace{1cm} (graph is sparse)
\hspace{1cm} (e.g. \( m \approx O(n) \))

---

**Basic Question:**
- Is there a path from \( s \) to \( t \)?
- Readable nodes from \( s \)?
- Is \( G \) connected?
- Find all connected components \( \text{of } G \)?
Basic Search Algorithms:
- Breadth First Search (BFS)
- Depth First Search (DFS)

**Example Trees**

**BFS**
- Nodes are explored in increasing order of their distance from the root.
- Visits nodes in a breadth-first manner, level by level.
- Order: 0 1 2 3 4 5 6 7 8 9 10

**DFS**
- Nodes are explored in a depth-first manner, exploring as far as possible along each branch before backtracking.
- Discovery order: 0 1 2 3 4 5 6 7 8 9 10
- Order: Pre-order: 0 1 2 3 4 5 6 7 8 9 10
- Order: Post-order: 0 1 2 3 4 5 6 7 8 9 10

**Graphs (Extension):**

- Main Idea: Do not revisit a vertex, if seen it before.

**BFS**
- Visits nodes in increasing order of their distance from the root.
- No cross edges.

**DFS**
- Visits nodes in a depth-first manner, exploring as far as possible along each branch before backtracking.
- Visits nodes in a depth-first manner, exploring as far as possible along each branch before backtracking.
- No cross edges.

**Non-tree edges:**
- **Back:** from a node to its ancestor.
- **Forward:** from a node to its successor.
- **Cross:** All other non-tree edges.

**Undirected:**
- Edge directions are possible.
- Not possible.
Implementation:

```c
BFS(G, s)
```

- **Idea 1:** Mark visited vertices.
- **Idea 2:** Use a queue data structure $Q$ ...

1. for each $u \in V$, do unmark $u$.
2. Insert $s$ in $Q$. Mark $s$. $\text{level}[s] = 0$
3. While $Q \neq \emptyset$
   4. remove a vertex $v$ from $Q$.
   5. for each $u \in \text{Adj}(v)$, do $\exists$
   6. if $v$ is unmarked.
   7. insert $v$ in $Q$. Mark $v$. at tail/end
   8. Parent [v] = u. $\text{level}[v] = \text{level}[u] + 1$

**Running Time:** $\text{lines 5-7} \leq O(1 + |\text{Adj}(v)|)$

Total time: $O(n + \sum_{u \in V} |\text{Adj}(u)|)$

$= O(n + m)$

**Global time = 0**

DFS $(G, s)$

// similar, or different data structure: stack
OR Recursion.

1. DFS-all $(G)$
OR Recursion.

1. Mark s.
2. for each \( v \in \text{Adj}(s) \) do
3. if \( v \) is unmarked.
4. \( \text{DFS}(G, v); \text{parent}[v] = s; \)
5. \( \text{finish}[s] = \text{time} + 1; \)

Running Time: \( O(n + m) \)

Applications:

Ex 1: Given undir/dir \( G, s,t \in V \\
find shortest path from \( s \) to \( t \).

1. Run BFS from \( s \).
2. level \( \text{ctJ} \) is the shortest path distance \( s \right\}_{t} \\
shortest path in BFS tree is the shortest path.

Ex 2: Undirected \( G \). Find all connected (components).

→ Run DFS-all \( (G) \)

→ Find DFS trees.
Ex 3: Given a directed graph G, decide if G has a cycle.

- Run DFS/BFS.
- Check if there is a re-visit edge.

Ex 4: Given a directed graph G, decide if G has a cycle.

- Run DFSAll(G).
- If there is a back edge \( \Rightarrow \) there is a cycle.

Proof of correctness:

\[ \exists \text{ a back edge} \Rightarrow \exists \text{ a cycle}. \]

\[ \exists \text{ a cycle } c \Rightarrow \exists \text{ a back edge}. \]

- Let \( u \) be the first vertex on \( c \) encountered by DFS.
- Let \( v \) be the vertex immediately before \( u \) on \( c \).

\( uv \) is a back edge in DFS.
is a back edge from $v$ to $u$. 