CHAPTER 3
GRAPH

Outline

- Content:
  - Basic definitions and applications
  - Graph connectivity and graph traversal
  - Implementation
  - Testing bipartiteness: an application of BFS
  - Connectivity in directed graphs
  - Directed acyclic graphs and topological ordering

- Reading:
  - Chapter 3

Keys to Success: CAR Theorem

Criticality
- Extract the essence
  - Identify the clean core
  - Remove extraneous detail

Abstraction
- Represent in an abstract form
  - First think at high-level
    - Devise the algorithm
  - Then go down to low-level
    - Complete implementation

Restriction
- Simplify unimportant things
  - List the limitations
  - Show how to extend

Definitions and applications

© Prof. Yao-Wen Chang@NTU
Salute to Euler!

Our focus in this course is on problems with a discrete flavor.

One of the most fundamental and expressive of combinatorial structures is the graph.

- Invented by L. Euler based on his proof on the Königsberg bridge problem (the seven bridge problem) in 1736.
  - Is it possible to walk across all the bridges exactly once and return to the starting land area?
  - Abstraction!

Graphs

Examples of Graphs (1/6)

- It's useful to digest the meaning of the nodes and the meaning of the edges in the following examples.
  - It's not important to remember them.

- Transportation networks:

Graphs

Examples of Graphs (2/6)

- Communication networks

Graphs
Examples of Graphs (3/6)

- Information networks
  - World Wide Web
    (node: webpage; edge: hyperlink)

Examples of Graphs (4/6)

- Social networks
  - Facebook
    (node: people; edge: friendship)

Examples of Graphs (5/6)

- Dependency networks
  - Food chain/web
    (node: species; edge: from prey to predator)

Examples of Graphs (6/6)

- Technological Network
  - Finite state machine
    (node: state; edge: state transition)
One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges.

- Browse Web pages by following hyperlinks
- Join a 10-day tour from Taipei to Europe on a sequence of flights
- Pass gossip by word of mouth (by message of mobile phone) from you to someone far away
  “Hey upper east siders, Gossip Girl here!”

-- Gossip Girl

A path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, ..., v_k \) with the property that each consecutive pair \( v_i, v_{i+1} \) is joined by an edge in \( E \).

- A path is simple if all nodes are distinct.
- A cycle is a path \( v_1, v_2, ..., v_k, v_1 \) in which \( v_1 = v_k \), \( k > 2 \), and the first \( k-1 \) nodes are all distinct.
- An undirected graph is connected if, for every pair of nodes \( u \) and \( v \), there is a path from \( u \) to \( v \).

The distance between nodes \( u \) and \( v \) is the minimum number of edges in a \( u-v \) path. (∞ for disconnected)

Note: These definitions carry over naturally to directed graphs with respect to the directionality of edges.

Path \( P = 1, 2, 4, 5, 3, 7, 8 \)
Cycle \( C = 1, 2, 4, 5, 3, 1 \)

An undirected graph is a tree if it is connected and does not contain a cycle.

- Trees are the simplest kind of connected graph: deleting any edge will disconnect it.

Thm: Let \( G \) be an undirected graph on \( n \) nodes. Any two of the following statements imply the third.

- \( G \) is connected.
- \( G \) does not contain a cycle.
- \( G \) has \( n-1 \) edges.

A rooted tree is a tree with its root at \( r \).

Rooted trees encode the notion of a hierarchy.

- e.g., sitemap of a Web site
  The tree-like structure facilitates navigation (root: entry page)
**Graph Connectivity and Graph Traversal**

**Node-to-Node Connectivity**

Q: Given a graph $G = (V, E)$ and two particular nodes $s$ and $t$, is there a path from $s$ to $t$ in $G$?
- The $s$-$t$ connectivity problem
- The maze-solving problem

A:
- For small graphs, easy! (visual inspection)
- 1-6 connectivity? 7-13 connectivity?
- What if large graphs? How efficiently can we do?

**Breadth-First-Search (BFS)**

- Breadth-first search (BFS): propagate the waves
  - Start at $s$ and flood the graph with an expanding wave that grows to visit all nodes that it can reach.
  - Layer $L_i$: $i$ is the time that a node is reached.
    - Adjacent nodes
    - Layer $L_0 = \{s\}$; layer $L_1$ = all neighbors of $L_0$.
    - Layer $L_{j+1}$ = all nodes that do not belong to an earlier layer and that are neighbors of $L_j$.

  $i$ = distance between $s$ to the nodes that belong to layer $L_j$.

**BFS Tree**

- Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$ respectively, and let $(x, y)$ be an edge of $G$. Then $i$ and $j$ differ by at most 1.
- Pf:
  - Without loss of generality, suppose $j - i > 1$.
  - By definition, $x \in L_i$, $x$'s neighbors belongs to $L_{i+1}$ or earlier.
  - Since $(x, y)$ is an edge of $G$, $y$ is $x$'s neighbor, $y \in L_j$ and $j \leq i + 1$.
- Nontree edge
- Tree edge
**Connected Component**

A connected component containing \( s \) is the set of nodes that are reachable from \( s \).
- Connected component containing node 1 is \{1, 2, 3, 4, 5, 6, 7, 8\}.
- There are three connected components.
  - The other two are \{9, 10\} and \{11, 12, 13\}.

**Color Fill**

Q: Given lime green pixel in an image, how to change color of entire blob of neighboring lime pixels to blue?

A: Model the image as a graph.
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

**Depth-First Search (DFS)**

Depth-first search (DFS): Go as deeply as possible or retreat
- Start from \( s \) and try the first edge leading out, and so on, until reach a dead end. Backtrack and repeat.
  - A mouse in a maze without the map.
- Another method for finding connected component

DFS\((u)\)
1. mark \( u \) as explored and add \( u \) to \( R \)
2. foreach edge \((u, v)\) incident to \( u \) do
3.   if \( v \) is not marked as explored then
4.     recursively invoke DFS\((v)\)
DFS Tree

Let $T$ be a DFS tree, let $x$ and $y$ be nodes in $T$, and let $(x, y)$ be a nontree edge. Then one of $x$ or $y$ is an ancestor of the other.

**Pf:**
- WLOG, suppose $x$ is reached first by DFS.
- When $(x, y)$ is examined during DFS($x$), it is not added to $T$ because $y$ is marked explored.
- Since $y$ is not marked as explored when DFS($x$) was first invoked, it is a node that was discovered between the invocation and end of the recursive call DFS($x$).
- $y$ is a descendant of $x$.

Summary: BFS and DFS

**Similarity:** BFS/DFS builds the connected component containing $s$.

**Difference:** BFS tree is flat/short; DFS tree is narrow/deep.

- What are the nontree edges in BFS/DFS?
- Q: How to produce all the connected components of a graph?
- A:

Representing Graphs

A graph $G = (V, E)$
- $|V|$ = the number of nodes = $n$
- $|E|$ = the number of edges = $m$
- Cardinality (size) of a set

Dense or sparse?
- For a connected graph, $n - 1 \leq m \leq \binom{n}{2} \leq n^2$
- Linear time = $O(m+n)$
- Why? It takes $O(m+n)$ to read the input
### Adjacency Matrix

Consider a graph $G = (V, E)$ with $n$ nodes, $V = \{1, ..., n\}$.

- The **adjacency matrix** of $G$ is an $n \times n$ matrix $A$ where
  - $A[u, v] = 1$ if $(u, v) \in E$;
  - $A[u, v] = 0$, otherwise.

- **Time:**
  - $\Theta(1)$ time for checking if $(u, v) \in E$.
  - $\Theta(n)$ time for finding out all neighbors of some $u \in V$.

- **Space:** $\Theta(n^2)$
  - What if sparse graphs?

### Adjacency List

The **adjacency list** of $G$ is an array $Adj[]$ of $n$ lists, one for each node represents its **neighbors**

- $Adj[u] = \text{a linked list of } \{v | (u, v) \in E\}$.

- **Time:** degree of $u$: number of neighbors
  - $\Theta(\deg(u))$ time for checking one edge or all neighbors of a node.

- **Space:** $O(n + m)$
  - What if sparse graphs?

### What is a Queue?

- A **queue** is a set of elements from which we extract elements in first-in, first-out (FIFO) order.
  - We select elements in the same order in which they were added.

- **Time:**
  - $Q(\deg(u))$ time for checking one edge or all neighbors of a node.

- **Space:** $O(n^2)$
  - What if sparse graphs?

### What is a Stack?

- A **stack** is a set of elements from which we extract elements in last-in, first-out (LIFO) order.
  - Each time we select an element, we choose the one that was added most recently.

- **Time:**
  - $\Theta(n)$ time for finding out all neighbors of some $u \in V$.

- **Space:** $O(n + m)$
  - What if sparse graphs?
### Implementing stacks and queues

- Implement queues and stacks by linked lists
- **Data Link**: top, Pop, Push
- **Linked Stack**: Advantage:
  - Variable sizes
  - Insert/delete in \(O(1)\)

### Implementing BFS

- Adjacency list is ideal for implementing BFS
- **Adjacency list**: \(L_i\)
  1. \(i = 0; L[0] = \{s\}; T = \{\}\;
  2. while \((L[i]\) is not empty) do
     3. \(L[i+1] = \{\}\;
     4. for each (node \(u \in L[i]\) do
        5. for each (edge \((u, v)\) incident to \(u\) do
           6. if (Discovered\(v\) = false) then
              7. Discovered\(v\) = true
              8. \(T = T + \{(u, v)\}\)
              9. \(L[i+1] = L[i+1] + \{v\}\)
     10. \(i++\)

### Implementing DFS

- We implement DFS by adjacency list
  - **Recursive Procedure**
  1. mark \(u\) as explored and add \(u\) to \(R\)
  2. for each edge \((u, v)\) incident to \(u\) do
  3. if (\(v\) is not marked as explored) then
  4. recursively invoke DFS\(v\)

### Alternative implementation of DFS
- Process each adjacency list in the reverse order

### Summary: Implementation

- **Graph**:
  - Adjacency matrix vs. Adjacency list
    - **Winner**
      - Faster to find an edge?: Matrix
      - Faster to find degree?: List
      - Faster to traverse the graph?: List
      - Storage for sparse graph?: List
      - Storage for dense graph?: Matrix
      - Edge insertion or deletion?: Matrix
      - Better for most applications?: List

- **Graph traversal**
  - BFS: queue (or stack)
  - DFS: stack
  - \(O(n+m)\) time
Application of BFS

Bipartite Graphs

A bipartite graph (bigraph) is a graph whose nodes can be partitioned into sets $X$ and $Y$ in such a way that every edge has one end in $X$ and the other end in $Y$.

- $X$ and $Y$ are two disjoint sets.
- No two nodes within the same set are adjacent.

Testing Bipartiteness

Color the nodes with blue and red (two-coloring)

Procedure:
1. Assume $G = (V, E)$ is connected.
   - Otherwise, we analyze connected components separately.
2. Pick any node $s \in V$ and color it red.
   - Anyway, $s$ must receive some color.
3. Color all the neighbors of $s$ blue.
4. Repeat coloring red/blue until the whole graph is colored.
5. Test bipartiteness: every edge has ends of opposite colors.

Is a Graph Bipartite?

Q: Given a graph $G$, is it bipartite?
A: Color the nodes with blue and red (two-coloring)

If a graph $G$ is bipartite, then it cannot contain an odd cycle.
Implementation: Testing Bipartiteness

Q: How to implement this procedure?

A: BFS! $O(m + n)$-time

We perform BFS from any $s$, coloring $s$ red, all of layer $L_1$ blue, ...

Even/odd-numbered layers red/blue

Insert the following statements after line 10 of BFS($s$) (p. 34)

10. $L[i+1] = L[i+1] + \{v\}$
10a. if ($(i+1)$ is even) then
10b. Color[$v$] = red
10c. else
10d. Color[$v$] = blue

Proof: Correctness (1/2)

Let $G$ be a connected graph and let $L_0, L_1, ...$ be the layers produced by BFS starting at node $s$. Then exactly one of the following holds.

1. No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
2. An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf: Case 1 is trivial.

Proof: Correctness (2/2)

Pf: (Case 2)

Suppose $(x, y)$ is an edge with $x, y$ in same layer $L_j$.

Let $z = \text{lca}(x, y) = \text{lowest common ancestor}$. Let $L_i$ be the layer containing $z$.

Consider the cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.

Its length is $1 + (j-i) + (j-i)$, which is odd.

Let $x \in L_0, y \in L_j,$ and $(x, y) \in E$. Then $i$ and $j$ differ by at most 1.

Connectivity in Directed Graphs
Recap: Directed Graphs

- In a directed graph: asymmetric relationships
  - Edges are directed, i.e., \( (u, v) \) \( \neq (v, u) \)
  - e.g., \( u \) knows \( v \) (celebrity), while \( v \) doesn't know \( u \).
  - Directionality is crucial.
- Representation: Adjacency list
  - Each node is associated with two lists, instead of one in an undirected graph.
- Graph search algorithms: BFS/DFS
  - Almost the same as undirected graphs
  - Again, directionality is crucial.

Strong Connectivity

- Nodes \( u \) and \( v \) are mutually reachable if there is a path from \( u \) to \( v \) and also a path from \( v \) to \( u \).
- A directed graph is strongly connected if every pair of nodes are mutually reachable.
- Q: When are there no mutually reachable nodes?
- Lemma: If \( u \) and \( v \) are mutually reachable, and \( v \) and \( w \) are mutually reachable, then \( u \) and \( w \) are mutually reachable.
- Simple but important!
- Pf:

Strong Component

- The strong component containing \( s \) in a directed graph is the maximal set of all \( v \) s.t. \( s \) and \( v \) are mutually reachable.
- a.k.a. strongly connected component
- Theorem: For any two nodes \( s \) and \( t \) in a directed graph, their strong components are either identical or disjoint.
- Q: When are they identical? When are they disjoint?
- Pf:
  - Identical if \( s \) and \( t \) are mutually reachable
    - \( s \to v, s \to t, v \to t \)
  - Disjoint if \( s \) and \( t \) are not mutually reachable
    - Proof by contradiction
Directed Acyclic Graphs

- **Q**: If an undirected graph has no cycles, then what’s it?
  - **A**: A tree (or forest).
    - At most \( n-1 \) edges.

- A **directed acyclic graph (DAG)** is a directed graph without cycles.
  - A DAG may have a rich structure.
  - A DAG encodes dependency or precedence constraints
    - e.g., prerequisite of Algorithms:
      - Data structures
      - Discrete math
      - Programming C/C++
    - e.g., execution order of instructions in CPU
    - Pipeline structures

**Example**

- **Q**: If \( G \) is a DAG, then does \( G \) have a topological ordering?
- **Q**: If so, how do we compute one?
  - **A**: Key: find a way to get started!
- **Q**: How?

**Topological Ordering**

- **Q**: 4 drivers come to the junction simultaneously, who goes first?
  - **A**: Deadlock! Dependencies form a cycle!
    - The driver must come to a complete stop at a stop sign. Generally the driver who arrives and stops first continues first. If two or three drivers in different directions stop simultaneously at a junction controlled by stop signs, generally the drivers on the left must yield the right-of-way to the driver on the far right.

- **Given a directed graph \( G \), a topological ordering is an ordering of its nodes as \( v_1, v_2, \ldots, v_n \) so that for every edge \((v_i, v_j)\), we have \( i < j \).**
  - **Precedence constraints**: edge \((v_i, v_j)\) means \( v_i \) must precede \( v_j \).

**Lemma**: If \( G \) has a topological ordering, then \( G \) is a DAG.

- **Pf**: Proof by contradiction!
  - **How?** Consider a cycle, \( v_i, \ldots, v_j, v_i \).

- **Example**
  - a DAG
  - the same DAG with topological ordering

**Lemma**: If \( G \) has a topological ordering, then \( G \) is a DAG.
Where to Start?

- A: A node that depends on no one, i.e., unconstrained.
- Lemma: In every DAG $G$, there is a node with no incoming edges.
  - Pf: Proof by contradiction!
    - Suppose that $G$ is a DAG where every node has at least one incoming edge. Let’s see how to find a cycle in $G$.
    - Pick any node $v$, and begin following edges backward from $v$: Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
    - Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$; and so on.
    - Repeat this process $n+1$ times (the initial $v$ counts one). We will visit some node $w$ twice, since $G$ has only $n$ nodes.
    - Let $C$ denote the sequence of nodes encountered between successive visits to $w$. Clearly, $C$ is a cycle.

Topological Ordering

- Lemma: If $G$ is a DAG, then $G$ has a topological ordering.
  - Pf: Proof by induction!
    1. Base case: true if $n = 1$.
    2. Inductive step:
       - Induction hypothesis: true for DAGs with up to $n$ nodes
       - Given a DAG on $n+1$ nodes, find a node $v$ w/o incoming edges.
         - $G - \{v\}$ is a DAG, since deleting $v$ cannot create any cycles.
         - $G - \{v\}$ has $n$ nodes. By induction hypothesis, $G - \{v\}$ has a topological ordering.
         - Place $v$ first in topological ordering. This is safe since all edges of $v$ point forward.
         - Then append nodes of $G - \{v\}$ in topological order after $v$.

A Linear-Time Algorithm

- TopologicalOrder($G$)
  1. find a node $v$ without incoming edges
  2. order $v$
  3. $G = G - \{v\}$ // delete $v$ from $G$
  4. if ($G$ is not empty) then TopologicalOrder($G$)

- Time: From $O(n^2)$ to $O(m+n)$
  - $O(n^2)$-time: Total $n$ iterations; line 1 in $O(n)$-time. How?
  - $O(m+n)$-time: How? Maintain the following information
    - indeg($w$) = # of incoming edges from undeleted nodes
    - $S$ = set of nodes without incoming edges from undeleted nodes
  - Initialization: $O(m+n)$ via single scan through graph
  - Update: line 3 deletes $v$
    - Remove $v$ from $S$
    - Decrement indeg($w$) for all edges from $v$ to $w$, and add $w$ to $S$ if indeg($w$) hits 0; this is $O(1)$ per edge