GRAPHS

COMP 321 – McGill University
These slides are mainly compiled from the following resources.
- Professor Jaehyun Park’ slides CS 97SI
- Top-coder tutorials.
- Programming Challenges books.
Outline

• Graphs.
  • Representation.
  • Special Graphs.
  • Graph Traversal.
  • Topological Sort.
  • Eulerian Circuit.
  • Minimum Spanning Tree.
  • Strongly Connected Components.

• Divide and Conquer.
Graphs

- An abstract way of representing connectivity using nodes (also called vertices) and edges.
- $m$ edges connect some pairs of nodes.
  - Edges can be either one-directional (directed) or bidirectional.
- Nodes and edges can have some auxiliary information.
Graphs

- Lots of problems formulated and solved in terms of graphs
  - Shortest path problems
  - Network flow problems
  - Matching problems
  - 2-SAT problem
  - Graph coloring problem
  - Traveling Salesman Problem (TSP): still unsolved!
  - and many more...
Graphs: Representation

- Need to store both the set of nodes V and the set of edges E
  - Nodes can be stored in an array
  - Edges must be stored in some other way
- Want to support operations such as:
  - Retrieving all edges incident to a particular node
  - Testing if given two nodes are directly connected
- Use either adjacency matrix or adjacency list to store the edges
Graphs: Adjacency Matrix

- An easy way to store connectivity information
  - Checking if two nodes are directly connected: O(1) time

- Make an n X n matrix A
  - \( a_{ij} = 1 \) if there is an edge from i to j
  - \( a_{ij} = 0 \) otherwise

- Uses O\( (n^2) \) memory
  - Only use when n is less than a few thousands,
  - and when the graph is dense
Graphs: Adjacency List

- Each node has a list of outgoing edges from it
  - Easy to iterate over edges incident to a certain node
  - The lists have variable lengths
  - Space usage: $O(n + m)$
Graphs: Implementing Adjacency List

• Solution 1. Using linked lists
  • Too much memory/time overhead
  • Using dynamic allocated memory or pointers is bad

• Solution 2. Using an array of vectors
  • Easier to code, no bad memory issues
  • But very slow

• Solution 3. Using arrays (!)
  • Assuming the total number of edges is known
  • Very fast and memory-efficient
Graphs: Implementing Adjacency List

- Solution 3. Using arrays (!)
  - Assuming the total number of edges is known
  - Very fast and memory-efficient

<table>
<thead>
<tr>
<th>ID</th>
<th>To</th>
<th>Next Edge ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Edge ID</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>
Graphs: Implementing Adjacency List

- Have two arrays E of size m and LE of size n
  - E contains the edges
  - LE contains the starting pointers of the edge lists
- Initialize LE[i] = -1 for all i
- Inserting a new edge from u to v with ID k
  - E[k].to = v
  - E[k].nextID = LE[u]
  - LE[u] = k
Graphs: Trees

• A connected acyclic graph
• Most important type of special graphs
  • Many problems are easier to solve on trees
• Alternate equivalent definitions:
  • A connected graph with \( n - 1 \) edges
  • An acyclic graph with \( n - 1 \) edges
  • There is exactly one path between every pair of nodes
  • An acyclic graph but adding any edge results in a cycle
  • A connected graph but removing any edge disconnects it
Graphs: Other Special Graphs

• Directed Acyclic Graph (DAG): the name says what it is
  • Equivalent to a partial ordering of nodes

• Bipartite Graph: Nodes can be separated into two groups S and T such that edges exist between S and T only (no edges within S or within T)
Graphs: Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- We will cover two algorithms
  - Depth-First Search (DFS): uses recursion (stack)
  - Breadth-First Search (BFS): uses queue
Graphs: Depth-First Search

- DFS(v): visits all the nodes reachable from v in depth-first order
  - Mark v as visited
  - For each edge v -> u:
    - If u is not visited, call DFS(u)
- Use non-recursive version if recursion depth is too big (over a few thousands)
  - Replace recursive calls with a stack
- dfs runs in O(V + E) and O(V^2) if the graph is stored as Adjacency List and Adjacency Matrix, respectively.
Graphs: Depth-First Search

- \( \text{dfs}(0) = 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4. \)
  - DS goes to the deepest possible vertex from the start vertex before attempting another branches.
- This sequence of visitation depends very much on the way we order neighbors of a vertex.
  - sequence 0 \( \rightarrow \) 1 \( \rightarrow \) 3 \( \rightarrow \) 2 (backtrack to 3) \( \rightarrow \) 4 is also a possible.
Graphs: Breadth-First Search

• BFS(v): visits all the nodes reachable from v in breadth-first order
  • Initialize a queue Q
  • Mark v as visited and push it to Q
  • While Q is not empty:
    • Take the front element of Q and call it w
    • For each edge w -> u :
      • If u is not visited, mark it as visited and push it to Q
Graphs: Breadth-First Search

- Source Vertex $s = 35$
# Graphs: DFS vs BFS

<table>
<thead>
<tr>
<th></th>
<th>(O(V + E)) DFS</th>
<th>(O(V + E)) BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pro</td>
<td>Uses less memory</td>
<td>Can solve SSSP on unweighted graphs</td>
</tr>
<tr>
<td>Cons</td>
<td>Cannot solve SSSP on unweighted graphs</td>
<td>Uses more memory</td>
</tr>
<tr>
<td>Code</td>
<td>Slightly easier to code</td>
<td>Slightly longer to code</td>
</tr>
</tbody>
</table>
Graphs: Topological Sort

• Input: a DAG G = (V,E)
• Output: an ordering of nodes such that for each edge u->v, u comes before v
• There can be many answers
  • e.g., both {6, 1, 3, 2, 7, 4, 5, 8} and {1, 6, 2, 3, 4, 5, 7, 8} are valid orderings for the graph below.
Graphs: Topological Sort

• Any node without an incoming edge can be the first element
• After deciding the first node, remove outgoing edges from it
• Repeat!

• Time complexity: \( O(n^2 + m) \)
  • Too slow...
Graphs: Topological Sort

• Precompute the number of incoming edges deg(v) for each node v
• Put all nodes v with deg(v) = 0 into a queue Q
• Repeat until Q becomes empty:
  • Take v from Q
  • For each edge v -> u:
    • Decrement deg(u) (essentially removing the edge v -> u)
    • If deg(u) = 0, push u to Q
• Time complexity: O(n + m)
Graphs: Eulerian Circuit

• Given an undirected graph G
• Want to find a sequence of nodes that visits every edge exactly once and comes back to the starting point

• Eulerian circuits exist if and only if
  • G is connected
  • and each node has an even degree
Graphs: Related Problems

- Eulerian path: exists if and only if the graph is connected and the number of nodes with odd degree is 2.
  - An Euler path starts and ends at different vertices.
- Hamiltonian path/cycle: a path/cycle that visits every node in the graph exactly once. Looks similar but very hard (still unsolved)!
Graphs: Eulerian Circuit

Euler circuit: CDEBBBADC

Euler path: CDCBBBADEEB
Graphs: Eulerian Circuit

<table>
<thead>
<tr>
<th># odd vertices</th>
<th>Euler path?</th>
<th>Euler circuit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No</td>
<td>Yes*</td>
</tr>
<tr>
<td>2</td>
<td>Yes*</td>
<td>No</td>
</tr>
<tr>
<td>4, 6, 8, ...</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>1, 3, 5,</td>
<td>No such graphs exist</td>
<td>No</td>
</tr>
</tbody>
</table>

* Provided the graph is connected.
Graphs: Eulerian Path

- Make sure the graph has either 0 or 2 odd vertices.
- If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
- Follow edges one at a time. If you have a choice between a bridge and a non-bridge, always choose the non-bridge.
- Stop when you run out of edges.
- This is called Fleury's algorithm,
Graphs: Eulerian Circuit

Problem: Find an Euler circuit in the graph below.
There are two odd vertices, A and F. Let’s start at F.
Start walking at F. When you use an edge, delete it.
Path so far: FE
Path so far: FEA
Path so far: FEAC
Path so far: FEACB
Up until this point, the choices didn’t matter.

But now, crossing the edge BA would be a mistake, because we would be stuck there.

The reason is that BA is a bridge. We don’t want to cross ("burn"?) a bridge unless it is the only edge available.
Path so far: FEACB
Path so far: FEACBD.
Path so far: FEACBD. Don’t cross the bridge!
Path so far: FEACBDC
Path so far: FEACBDC  Now we have to cross the bridge CF.
Path so far: FEACBDFC
Path so far: FEACBDCFD
Path so far: FEACBDBD
Euler Path: FEACBDCFDBA
Graphs: Minimum Spanning Tree

- Given an undirected weighted graph $G = (V,E)$
- Want to find a subset of $E$ with the minimum total weight that connects all the nodes into a tree

- Two famous algorithms:
  - Kruskal’s algorithm
  - Prim’s algorithm
Graphs: Kruskal’s Algorithm

• Main idea: the edge e⋆ with the smallest weight has to be in the MST.

• Another main idea: after an edge is chosen, the two nodes at the ends can be merged and considered as a single node (supernode).

• Pseudocode:
  • Sort the edges in increasing order of weight
  • Repeat until there is one supernode left:
    • Take the minimum weight edge e⋆
    • If e⋆ connects two different supernodes, then connect them and merge the supernodes (use union-find)
    • Otherwise, ignore e⋆ and try the next edge
Graphs: Kruskal’s Algorithm
Graphs: Prim’s Algorithm

- Main idea:
  - Maintain a set $S$ that starts out with a single node $s$
  - Find the smallest weighted edge $e^* = (u, v)$ that connects $u \in S$ and $v \notin S$
  - Add $e^*$ to the MST, add $v$ to $S$
  - Repeat until $S = V$

- Differs from Kruskal’s in that we grow a single supernode $S$ instead of growing multiple ones at the same time
Graphs: Prim’s Algorithm
Graphs: Prim’s Algorithm

• Initialize $S := \{s\}$, $D_v := \text{cost}(s,v)$ for every $v$
  • If there is no edge between $s$ and $v$, $\text{cost}(s,v) = 1$
• Repeat until $S = V$:
  • Find $v \notin S$ with smallest $D_v$
    • Use a priority queue or a simple linear search
  • Add $v$ to $S$, add $D_v$ to the total weight of the MST
  • For each edge $(v, w)$:
    • Update $D_w := \min(D_w, \text{cost}(v, w))$
Graphs: Kruskal’s VS Prim’s

• Kruskal’s Algorithm
  • Takes $O(m \log(m))$ time
  • Pretty easy to code
  • Generally slower than Prim’s

• Prim’s Algorithm
  • Time complexity depends on the implementation:
    • Can be $O(n^2 + m)$, $O(m \log(n))$, or $O(m + n \log n)$
    • A bit trickier to code
    • Generally faster than Kruskal’s
Graphs: Strongly Connected Components

- Given a directed graph \( G = (V,E) \)
- A graph is strongly connected if all nodes are reachable from every single node in \( V \)
- Strongly connected components of \( G \) are maximal strongly connected subgraphs of \( G \)
- The graph below has 3 SCCs: \{a, b, e\}, \{c, d, h\}, \{f, g\}

![Graph Diagram](https://via.placeholder.com/150)

Figure from Wikipedia
Graphs: Kosaraju’s Algorithm

- Initialize counter c := 0
- While not all nodes are labeled:
  - Choose an arbitrary unlabeled node v
  - Start DFS from v
    - Check the current node x as visited
    - Recurse on all unvisited neighbors
    - After the DFS calls are finished, increment c and set the label of x as c
- Reverse the direction of all the edges
- For node v with label n, n – 1, . . . , 1:
  - Find all reachable nodes from v and group them as an SCC