Graphs

Graphs are not function plots.

What is a Graph?

- A graph $G$ is a pair $(V,E)$
  where $V$: set of vertices
  $E$: set of edges connecting the vertices in $V$
  Vertices and edges are positions and store elements
- An edge $e = (u,v)$ is a pair of vertices

Example:

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

*Graph is a more general data structure. Tree is a special case of graph*
Graph: Example
- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route

Graph: Examples
- Computer networks
  - Local area network
  - Internet
  - Web
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Databases
  - Entity-relationship diagram

Edge Types
- Directed edge
  - ordered pair of vertices \((u, v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices \((u, v)\)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network
Graph Terminology

- **adjacent vertices**: vertices connected by an edge
- **degree (of a vertex)**: # of adjacent vertices

**NOTE**: The sum of the degrees of all vertices is twice the number of edges. Why?

Since adjacent vertices each count the adjoining edge, it will be counted twice.

- **path**: sequence of vertices $v_1, v_2, \ldots, v_k$ such that consecutive vertices $v_i$ and $v_{i+1}$ are adjacent.

More Graph Terminology

- **simple path**: no repeated vertices

- **cycle**: simple path, except that the last vertex is the same as the first vertex

Subgraphs

- A **subgraph** $S$ of a graph $G$ is a graph such that
  - The vertices of $S$ are a subset of the vertices of $G$
  - The edges of $S$ are a subset of the edges of $G$

- A **spanning subgraph** of $G$ is a subgraph that contains all the vertices of $G$
Connectivity

- A graph is **connected** if there is a path between every pair of vertices.
- A **connected component** of a graph $G$ is a connected subgraph of $G$.

Trees and Forests

- A **forest** is an undirected graph without cycles.
- The connected components of a forest are **trees**.
- A (free) **tree** is an undirected graph $T$ such that
  - $T$ is connected
  - $T$ has no cycles
  - This definition of tree is different from the one of a rooted tree.
Trees and Forests

Let \( n = \text{#vertices}, \) and \( m = \text{#edges} \)

How many total edges in a tree? \[ m = n - 1 \]

Total edges in a forest? \[ m < n - 1 \]

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- A spanning forest of a graph is a spanning subgraph that is a forest

Data Structures for Graphs

- A Graph! How can we represent it?
  - Edge list
  - Adjacency lists
  - Adjacency matrix
**Edge List**

- The edge list structure simply stores the vertices and the edges into two containers (e.g., lists, vectors etc.).
- Each edge object has references to the vertices it connects.

Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence. Time: $O(m)$

**Adjacency List (traditional)**

- adjacency list of a vertex $v$: sequence of vertices adjacent to $v$
- represent the graph by the adjacency lists of all the vertices

$$\text{Space} = \Theta(n + \sum \deg(v)) = \Theta(n + m)$$

**Adjacency List (modern)**

- The adjacency list structure extends the edge list structure by adding incidence containers to each vertex.

$$\text{Space is } O(n + m)$$
Performance of the Adjacency List Structure

- size, isEmpty, replaceElement, swap: \(O(1)\)
- numVertices, numEdges: \(O(1)\)
- vertices: \(O(n)\)
- edges, directedEdges, undirectedEdges: \(O(m)\)
- elements, positions: \(O(n+m)\)
- endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree: \(O(1)\)
- incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v): \(O(deg(v))\)
- outAdjacentVertices(v): \(O(min(deg(u), deg(v)))\)
- insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection: \(O(1)\)
- removeVertex(v): \(O(deg(v))\)

Adjacency Matrix (traditional)

- matrix \(M\) with entries for all pairs of vertices
- \(M_{i,j} = \text{true}\) means that there is an edge \((i,j)\) in the graph.
- \(M_{i,j} = \text{false}\) means that there is no edge \((i,j)\) in the graph.
- There is an entry for every possible edge, therefore:
  
  \[
  \text{Space} = \Theta(n^2)
  \]

Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.
Data Structures for Graphs

- A Graph! How can we represent it?
  - Edge list
  - Adjacency lists
  - Adjacency matrix

Edge List

- The edge list structure simply stores the vertices and the edges into two containers (e.g., lists, vectors etc.)
- Each edge object has references to the vertices it connects.

Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence. Time: O(m)

Adjacency List

- The adjacency list structure extends the edge list structure by adding incidence containers to each vertex.

Space is \(O(n + m)\).
Adjacency Matrix

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>AA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TW</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>AA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Graph Traversal

- A procedure for exploring a graph by examining all of its vertices and edges.
- Two different techniques:
  - Depth First traversal (DFT)
  - Breadth First Traversal (BFT)

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph.
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
DFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges.

**Algorithm DFS(G)**
- **Input**: graph G
- **Output**: labeling of the edges of G as discovery edges and back edges

```plaintext
for all u in G.vertices():
    setLabel(u, UNEXPLORED)
for all e in G.edges():
    setLabel(e, UNEXPLORED)
for all v in G.vertices():
    if getLabel(v) == UNEXPLORED:
        DFS(G, v)
```

**Algorithm DFS(G, v)**
- **Input**: graph G and a start vertex v of G
- **Output**: labeling of the edges of G in the connected component of v as discovery edges and back edges

```plaintext
setLabel(v, VISITED)
for all e in G.incidentEdges(v):
    if getLabel(e) == UNEXPLORED:
        w ← opposite(v, e)
        if getLabel(w) == UNEXPLORED:
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else:
            setLabel(e, BACK)
```

Example

Example (cont.)

DFS Algorithm
Properties of DFS

Property 1:
\( \text{DFS}(G, v) \) visits all the vertices and edges in the connected component of \( v \).

Analysis of DFS

- Setting/getting a vertex/edge label takes \( O(1) \) time.
- Each vertex is labeled twice:
  - once as 'UNEXPLORED'
  - once as 'VISITED'
- Each edge is labeled twice:
  - once as 'UNEXPLORED'
  - once as 'DISCOVERY' or 'BACK'
- DFS runs in \( O(n + m) \) time provided the graph is represented by the adjacency list structure.

Depth-First Search

- DFS on a graph with \( n \) vertices and \( m \) edges takes \( O(n + m) \) time.
- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices
  - Find a cycle in the graph

Analysis of DFS

- DFS on a graph with \( n \) vertices and \( m \) edges takes \( O(n + m) \) time.
- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices
  - Find a cycle in the graph

Depth-First Search
Path Finding

- We call $DFS(G, u)$ with $u$ as the start vertex.
- We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex $v$ is encountered, we return the path as the contents of the stack.

Algorithm: pathDFS($G, v, z$)

1. setLabel($v$, VISITED)
2. S.push($v$)
3. if $v = z$
   - return S.elements()
4. for all $e \in G.incidentEdges(v)$
   - if getLabel($e$) = UNEXPLORED
     - $w \leftarrow$ opposite($v, e$)
     - if getLabel($w$) = UNEXPLORED
       - setLabel($e$, DISCOVERY)
       - S.push($e$)
       - pathDFS($G, w, z$)
       - S.pop($e$)
     - else
       - setLabel($e$, BACK)
       - S.pop($v$)

Cycle Finding

- We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
- As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$. 
Cycle Finding

Algorithm cycleDFS(G, v, z)
setLabel(v, VISITED)
S.push(v)
for all e ∈ G.incidentEdges(v)
if getLabel(e) = UNEXPLORED
w ← opposite(e)
S.push(e)
if getLabel(w) = UNEXPLORED
setLabel(e, DISCOVERY)
pathDFS(G, w, z)
S.pop(v)

Review: Representation

Space = \( \Theta(n + \Sigma \text{deg}(v)) = \Theta(n + m) \)

Review: DFS
Review: DFS
Review: DFS

1. a
2. b
3. c
4. d
5. e

pa-b
pa-c
pa-d
pa-b
pb-e
pa-c
pc-d
pc-e
pa-d
pc-d
pd-e
pb-e
pc-e
pd-e

a-b
pa | pb
a-c
pa | pc
a-d
pa | pd
b-e
pb | pe
c-d
pc | pd
c-e
pc | pe
d-e
pd | pe
Review: DFS

Breadth-First Search
BFS Algorithm

The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

Algorithm BFS(G, s)
Input graph G
Output labeling of the edges and partition of the vertices of G
for all u ∈ G.vertices()
setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
if getLabel(v) = UNEXPLORED
BFS(G, v)

Properties

Notation
G_s: connected component of s

Property 1
BFS(G, s) visits all the vertices and edges of G_s

Property 2
The discovery edges labeled by BFS(G, s) form a spanning tree T_s of G_s

Analysis

• Setting/getting a vertex/edge label takes \(O(1)\) time
• Each vertex is labeled twice
  – once as UNEXPLORED
  – once as VISITED
• Each edge is labeled twice
  – once as UNEXPLORED
  – once as DISCOVERY or CROSS
• Each vertex is inserted once into a queue \(L\)
• Method incidentEdges is called once for each vertex
• BFS runs in \(O(n + m)\) time provided the graph is represented by the adjacency list structure
Applications

- we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Back edge $(v,w)$
- $w$ is an ancestor of $v$ in the tree of discovery edges

Cross edge $(v,w)$
- $w$ is in the same level as $v$ or in the next level in the tree of discovery edges

DFS vs. BFS (cont.)

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Shortest paths</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

Biconnected components

Shortest paths

Spanning forest, connected components, paths, cycles
Quiz 10

1. Let $G$ be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table. Assume that, in a traversal of $G$, the adjacent vertices of a given vertex are returned in the same order as they are listed in the above table.

(a) Draw $G$

(b) Order the vertices as they are visited in a DFS traversal starting at vertex 1.

(c) Order the vertices as they are visited in a BFS traversal starting at vertex 1.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 3, 4)</td>
</tr>
<tr>
<td>3</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>4</td>
<td>(1, 2, 3, 6)</td>
</tr>
<tr>
<td>5</td>
<td>(6, 7, 8)</td>
</tr>
<tr>
<td>6</td>
<td>(4, 5, 7)</td>
</tr>
<tr>
<td>7</td>
<td>(5, 6, 8)</td>
</tr>
<tr>
<td>8</td>
<td>(5, 7)</td>
</tr>
</tbody>
</table>

Quiz 10 (Contd.)

2. Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

(a) The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

(b) The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

(c) You need to answer the query `areAdjacent` as fast as possible no matter how much space you use.

Directed Graphs
Digraphs

- A **digraph** is a graph whose edges are all directed
  - Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling

Digraph Properties

- A graph $G=(V,E)$ such that
  - Each edge goes in one direction:
    - Edge $(a,b)$ goes from $a$ to $b$, but not $b$ to $a$.
  - If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.

DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices such that for every edge $(v_i, v_j)$, we have $i < j$
- A digraph admits a topological ordering if and only if it is a DAG

Topological ordering of $G$
Topological Sorting

- Number vertices, so that (u,v) in E implies u < v

**Strategy:** Pick up a vertex which has zero incident edges.
Graph with # of Incident edges for each vertex.
Topological Sorting Example

---

Topological Sorting Example

---

Topological Sorting Example

---
Algorithm for Topological Sorting

Algorithm TopologicalSort(G)
Let S be an empty stack
for each vertex u of G do
    set its in_counter
    if in_counter = 0 then
        Push u in S
    i ← 1
while S is not empty do
    Pop v from S
    Label v ← i
    i ← i + 1
for every w adjacent to v do
    reduce the in_counter of w by 1
    if in_counter = 0 then
        Push w in S
if (i < # of vertices)
    “Digraph has a directed cycle”
Run Time: O(n+m)
Space use: O(n)
Programming Assignment 3

- Implement a weighted graph ADT using adjacency list data structure with the following methods.
  - DFS
  - BFS
  - Dijkstra’s Shortest Path finding algorithm
  - Kruskal’s Minimum Spanning Tree algorithm