Graphs

CS16: Introduction to Data Structures & Algorithms
Spring 2019
Outline

‣ What is a Graph
‣ Terminology
‣ Properties
‣ Graph Types
‣ Representations
‣ Performance
‣ BFS/DFS
‣ Applications
What is a Graph

- A graph is defined by
  - a set of vertices \( V \)
  - a set of edges \( E \)
- Vertices and edges can both store data
- Example
  - vertices represent artists & can store content
  - edges represent collaborations (vocals or production)
A Graph

O(1)

Wu-Tang
Terminology

- Endpoints or end vertices of an edge
  - U and V are endpoints of edge a
- Incident edges of a vertex
  - a, b, d are incident to V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree of 5
- Parallel (multiple) edges
  - h, i are parallel edges
- Self-loops
  - j is a self-looped edge
Terminology

- A path is a sequence of alternating vertices and edges
  - begins and ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are visited at most once
- Examples
  - $P_1 = V \rightarrow_b X \rightarrow_h Z$ is a simple path
  - $P_2 = U \rightarrow_c W \rightarrow_e X \rightarrow_g Y \rightarrow_f W \rightarrow_d V$ is not a simple path, but is still a path
Applications

- Flight networks
- Road networks & GPS
- The Web
  - pages are vertices
  - links are edges
- The Internet
  - routers and devices are vertices
  - network connections are edges
- Facebook
  - profiles are vertices
  - friendships are edges
Graph Properties

- A graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

- A graph is **connected** if there exists a path from each vertex to every other vertex.

- A path is a **cycle** if it starts and ends at the same vertex.

- A graph is **acyclic** if it has no cycles.
A Subgraph
A Connected Graph
An Unconnected Graph

2 Connected Components
An Unconnected Graph

4 Connected Components
Cycles

[Diagram with interconnected images and labels]

O(1)

WU-TANG

[Image of Wu-Tang Clan]

O(1)

[Image of different artists and celebrities]
An Acyclic Graph
Graph Properties

- A **spanning tree** of $G$ is a subgraph with
  - all of $G$’s vertices in a single tree
  - and enough edges to connect each vertex w/o cycles
Spanning Tree

No cycles!
Graph Properties

- A **spanning forest** is
  - a subgraph that consists of a spanning tree in each connected component of graph

- Spanning forests never contain cycles
  - this might not be the “best” or shortest path to each node
Graph Properties

- $G$ is a tree if and only if it satisfies any of these conditions
  - $G$ has $|V| - 1$ edges and no cycles
  - $G$ has $|V| - 1$ edges and is connected
  - $G$ is connected, but removing any edge disconnects it
  - $G$ is acyclic, but adding any edges creates a cycle
  - Exactly one simple path connects each pair of vertices in $G$
Graph Proof 1

Prove that

the sum of the degrees of all vertices of some graph \( G \) is twice the number of edges of \( G \)

Let \( V = \{v_1, v_2, \ldots, v_p\} \), where \( p \) is number of vertices

The total sum of degrees \( D \) is such that

\[ D = \deg(v_1) + \deg(v_2) + \ldots + \deg(v_p) \]

But each edge is counted twice in \( D \)

\( \text{one for each of the two vertices incident to the edge} \)

So \( D = 2|E| \), where \( |E| \) is the number of edges.
Graph Proof 2

- Prove using induction that if $G$ is connected then
  - $|E| \geq |V| - 1$, for all $|V| \geq 1$

- Base case $|V| = 1$
  - If graph has one vertex then it will have 0 edges
  - so since $|E| = 0$ and $|V| - 1 = 1 - 1 = 0$, we have $|E| \geq |V| - 1$

- Inductive hypothesis
  - If graph has $|V| = k$ vertices then $|E| \geq k - 1$

- Inductive step
  - Let $G$ be any connected graph with $|V| = k + 1$ vertices
  - We must show that $|E| \geq k$
Graph Proof 2

- Inductive step
  - Let G be any connected graph with $|V| = k + 1$ vertices
  - We must show that $|E| \geq k$
- Let u be the vertex of minimum degree in G
  - $\deg(u) \geq 1$ since G is connected
- If $\deg(u) = 1$
  - Let G’ be G without u and its 1 incident edge
  - G’ has k vertices because we removed 1 vertex from G
  - G’ is still connected because we only removed a leaf
  - So by inductive hypothesis, G’ has at least $k-1$ edges
  - which means that G has at least k edges
Graph Proof 2

- If $\deg(u) \geq 2$
  - Every vertex has at least two incident edges
  - So the total degree $D$ of the graph is $D \geq 2(k+1)$
  - But we know from the last proof that $D=2|E|$
    - so $2|E| \geq 2(k+1) \implies |E| \geq k+1 \implies |E| \geq k$
- We showed it is true for $|V|=1$ (base case)...
  - ...and for $|V|=k+1$ assuming it is true for $|V|=k$...
  - ...so it is true for all $|V| \geq 1$
Edge Types

- **Undirected edge**
  - unordered pair of vertices (LH,CS)
  - for example a collaboration

- **Directed edge**
  - ordered pair of vertices (LH,CS)
  - first vertex LH is the origin
  - second vertex CS is the destination
  - LH collaborated with CS on CS’s album
An Undirected Graph
Directed Acyclic Graph (DAG)

Directed means ‘is a prerequisite for’

We’ll talk much more about DAGs in future lectures…

Acyclic = without cycles
Graph Representations

- Vertices usually stored in a List or Set
- 3 common ways of representing which vertices are adjacent
  - Edge list (or set)
  - Adjacency lists (or sets)
  - Adjacency matrix
Edge List

- Represents adjacencies as a list of pairs
- Each element of list is a single edge $(a, b)$
- Since the order of list doesn’t matter
  - can use hashset to improve runtime of adjacency testing
Edge Set

- Store all the edges in a HashSet

(1,1) (3,4) (2,5) (1,1) (1,5)
(4,6) (2,5) (4,5) (1,2)
(2,3)
Adjacency Lists

- Each vertex has an associated list with its neighbors
- Since the order of elements in lists doesn’t matter
  - lists can be hashsets instead
Adjacency Set

- Each vertex associated with a hashset of its neighbors.
Adjacency Matrix

- Matrix with \( n \) rows and \( n \) columns
  - \( n \) is number of vertices
  - If \( u \) is adjacent to \( v \) then \( M[u,v] = T \)
  - If \( u \) is not adjacent to \( v \) then \( M[u,v] = F \)
  - If graph is undirected then \( M[u,v] = M[v,u] \)
Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Adjacency Matrix

- Initialize matrix to predicted size of graph
  - we can always expand later
- When vertex is added to graph
  - reserve a row and column of matrix for that vertex
- When vertex is removed
  - set its entire row and column to false
- Since we can’t remove rows/columns from arrays
  - keep separate collection of vertices that are actually present in graph
Graph ADT

- Vertices and edges store values
  - Ex: edge weights
- Accessor methods
  - `vertices()`
  - `edges()`
  - `incidentEdges(vertex)`
  - `areAdjacent(v₁, v₂)`
  - `endVertices(edge)`
  - `opposite(vertex, edge)`
- Update methods
  - `insertVertex(value)`
  - `insertEdge(v₁, v₂)`
    - sometimes this function also takes a value
      so `insertEdge(v₁, v₂, val)`
  - `removeVertex(vertex)`
  - `removeEdge(edge)`
Big-O Performance

Activity #1

3 min
Big-O Performance

Activity #1

3 min
Big-O Performance

Activity #1

2 min
Big-O Performance

Activity #1

1 min
Big-O Performance

Activity #1
## Big-O Performance

<table>
<thead>
<tr>
<th></th>
<th>Edge Set</th>
<th>Adjacency Sets</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Space</td>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>vertices( )</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
</tr>
<tr>
<td>edges( )</td>
<td>$O(1)^*$</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>areAdjacent (v₁, v₂)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>insertEdge(v₁, v₂)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>removeEdge(v₁, v₂)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

1 In all approaches, we maintain an additional list or set of vertices

* in place (return pointer)
## Big-O Performance (Edge Set)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>$O(1)$</td>
<td>Return set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>$O(1)$</td>
<td>Return set of edges</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>areAdjacent(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Check if $(v₁,v₂)$ exists in the set</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>$O(1)$</td>
<td>Add vertex $v$ to the vertex list</td>
</tr>
<tr>
<td>insertEdge(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Add element $(v₁,v₂)$ to the set</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>removeEdge(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Remove edge $(v₁,v₂)$</td>
</tr>
</tbody>
</table>
## Big-O Performance (Adjacency Set)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>$O(1)$</td>
<td>Return the set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>$O(1)$</td>
<td>Return v’s edge set</td>
</tr>
<tr>
<td>areAdjacent(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Check if v₂ is in v₁’s set</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>$O(1)$</td>
<td>Add vertex v to the vertex set</td>
</tr>
<tr>
<td>insertEdge(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Add v₁ to v₂’s edge set and vice versa</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>removeEdge(v₁,v₂)</td>
<td>$O(1)$</td>
<td>Remove v₁ from v₂’s set and vice versa</td>
</tr>
</tbody>
</table>
## Big-O Performance (Adjacency Matrix)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>(O(1))</td>
<td>Return the set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: row/col are the same in an undirected graph.</td>
</tr>
<tr>
<td>areAdjacent((v_1,v_2))</td>
<td>(O(1))</td>
<td>Check index ((v_1,v_2)) for a true</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td>insertEdge((v_1,v_2))</td>
<td>(O(1))</td>
<td>Set index ((v_1,v_2)) to true</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td>removeEdge((v_1,v_2))</td>
<td>(O(1))</td>
<td>Set index ((v_1,v_2)) to false</td>
</tr>
</tbody>
</table>
BFT and DFT

- Remember BFT and DFT on trees?
- We can also do them on graphs
  - a tree is just a special kind of graph
  - often used to find certain values in graphs
BFT/DFT on Graphs

Activity #2
BFT/DFT on Graphs

1 min

Activity #2
BFT/DFT on Graphs

Activity #2
Breadth First Traversal: Tree vs. Graph

function `treeBFT`(root):
  //Input: Root node of tree
  //Output: Nothing
  Q = new Queue()
  Q.enqueue(root)
  while Q is not empty:
    node = Q.dequeue()
    doSomething(node)
    enqueue node’s children

doSomething() could print, add to list, decorate node etc…

function `graphBFT`(start):
  //Input: start vertex
  //Output: Nothing
  Q = new Queue()
  start.visited = true
  Q.enqueue(start)
  while Q is not empty:
    node = Q.dequeue()
    doSomething(node)
    for neighbor in adj nodes:
      if not neighbor.visited:
        neighbor.visited = true
        Q.enqueue(neighbor)

Mark nodes as visited otherwise you will loop forever!
Depth First Traversal

- To do DFT on graph, replace queue with stack
- Can also be done recursively

```python
function recursiveDFT(node):
    // Input: start node
    // Output: Nothing
    node.visited = true
    for neighbor in node’s adjacent vertices:
        if not neighbor.visited:
            recursiveDFT(neighbor)
```
Applications: Flight Paths Exist

- Given undirected graph with airports & flights
  - is it possible to fly from one airport to another?
- Strategy
  - use breadth first search starting at first node
  - and determine if ending airport is ever visited
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?
Applications: Flight Paths Exist

- Is there a flight from SFO to PVD?

![Network Diagram]

- PWM
- JFK
- SFO
- ORD
- PVD
- HNL
- LAX
- DFW
- LGA
- MIA
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?

- Yes! but how do we do it with code?
Function `pathExists(from, to):`

// Input: from: vertex, to: vertex
// Output: true if path exists, false otherwise

Q = new Queue()
from.visited = true
Q.enqueue(from)

while Q is not empty:
    airport = Q.dequeue()
    if airport == to:
        return true
    for neighbor in airport’s adjacent nodes:
        if not neighbor.visited:
            neighbor.visited = true
            Q.enqueue(neighbor)

return false
Applications: Flight Layovers

- Given undirected graph with airports & flights
  - decorate vertices w/ least number of stops from a given source
  - if no way to get to an airport decorate w/ \( \infty \)

- Strategy
  - decorate each node w/ initial ‘stop value’ of \( \infty \)
  - use breadth first search to decorate each node…
  - …w/ ‘stop value’ of one greater than its previous value
function numStops(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest number of layovers from source.

    for every node in G:
        node.stops = infinity

    Q = new Queue()
    source.stops = 0
    source.visited = true
    Q.enqueue(source)
    while Q is not empty:
        airport = Q.dequeue()
        for neighbor in airport’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                neighbor.stops = airport.stops + 1
                Q.enqueue(neighbor)