CS 2110
Lecture 13

Graphs
• Terminology
• Graph ADT
• Adjacency lists
• Adjacency matrices
A4 Due Weekend

A5 Released

Today

A3 grades (ETA sometime today)
Bag  List  Graph  Tree  Stack  Queue  Dictionary
Charts (aka graphs)
Graphs

- **Graph**:  
  - [charts] Points connected by curves  
  - [in CS] Vertices connected by edges

- Graphs generalize trees

- Connections are important in real world programming problems…
Vertices: people
Edges: friendships

https://medium.com/@johnrobb/facebook-the-complete-social-graph-b58157ee6594
Vertices: stations
Edges: cables

https://www.submarinecablemap.com/
Vertices: State capitals
Edges: “States have shared border”
Vertices: maze junctions
Edges: maze corridors

Graphs as Mathematical Structures
Undirected graphs

An **undirected** graph is a pair \((V, E)\) where

- \(V\) is a set
  - Element of \(V\) is called a **vertex** or **node**
  - We’ll consider only finite graphs
  - Ex: \(V = \{A, B, C, D, F\}; \ |V| = 5\)

- \(E\) is a set
  - Element of \(E\) is called an **edge**
  - An edge is itself a two-element set \(\{u, v\}\) where \(\{u, v\} \subseteq V\)
  - Sometimes require \(u \neq v\) (i.e., no **self-loops**)
  - Ex: \(E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}, \ |E| = 4\)
Directed graphs

A directed graph is similar except the edges are pairs \((u, v)\), hence order matters.

\[ V = \{A, B, C, D, F\} \]
\[ E = \{(A, C), (B, A), (B, C), (C, D), (D, C)\} \]
\[ |V| = 5 \]
\[ |E| = 5 \]
• A path is a sequence $v_0, v_1, v_2, ..., v_p$ of vertices such that for $0 \leq i < p$,
  • Directed: $(v_i, v_{i+1}) \in E$
  • Undirected: $\{v_i, v_{i+1}\} \in E$

• The length of a path is its number of edges

The path ACD has length 2
Convert undirected to directed?

**Can do it:** Replace each edge \( \{u,v\} \) with two edges \( \{(u,v), (v,u)\} \)
Convert directed to undirected?

Can’t do it while also maintaining paths:

\[ \text{BA is not a path} \]

\[ \text{BA is a path} \]
Nodes and edges can optionally be labelled with any kind of data. When labelled with a number, called a **weight**. Weight of a path is the sum of weights on its edges.

These nodes could represent cities, and the integer edge labels could represent the distance in miles between cities.
Question

What is the length of the shortest path from Provincetown to Chatham?

A. 3
B. 4
C. 36
D. 69
E. 72
What is the length of the shortest path from Provincetown to Chatham?

- 3: 0%
- 4: 0%
- 36: 0%
- 69: 0%
- 72: 0%
Question

What is the weight of the shortest path from Provincetown to Chatham?

A. 3
B. 4
C. 36
D. 69
E. 72
What is the weight of the shortest path from Provincetown to Chatham?

- 3: 0%
- 4: 0%
- 36: 0%
- 69: 0%
- 72: 0%
Graphs as Data Structures
Graph ADT

Operations could include:

• Add a vertex
• Remove a vertex
• Search for a vertex
• Number of vertices
• Add an edge
• Remove an edge
• Search for an edge
• Number of edges
Code Demo: Graph Interface
Graph data structures

• Vertex u is adjacent to vertex v if there is an edge from v to u [sic]
  • “The vertices adjacent to v are the vertices you can reach from v by following one edge”

• Common graph data structures:
  • Adjacency list
  • Adjacency matrix

running example (directed, no edge labels)

2 is adjacent to 3
3 is not adjacent to 2
Question

Which of these is true?

A. cp4 is adjacent to cp2
B. cp2 is adjacent to cp4
C. cp6 is adjacent to cp2
D. Both A and B
E. Both A and C
Which of these is true?

- cs4 is adjacent to cs2 0%
- cs2 is adjacent to cs4 0%
- cs6 is adjacent to cs2 0%
- Both A and B 0%
- Both A and C 0%
Adjacency “list”

• Maintain a collection of the vertices
• For each vertex, also maintain a collection of its adjacent vertices

• Vertices: 1, 2, 3, 4
• Adjacencies:
  • 1: 2, 3
  • 2: 4
  • 3: 2, 4
  • 4: none

The vertices adjacent to vertex 1 are vertex 2 and vertex 3.
Adjacency “list” conceptual representation

- Vertices:
  - Set of vertex labels
  - `Set<Integer>` for example graph

- Adjacencies:
  - Dictionary mapping from vertex labels to sets of vertex labels
  - `Map<Integer, Set<Integer>>` for example graph

- But the reason for the name “list” is you *can* do it all with just linked lists…
Adjacency list implementation #1

**Linked list**, where each node contains vertex label and linked list of adjacent vertex labels
Adjacency list implementation #1

Or if you want edge labels:

running example (directed, with edge labels)
Code Demo: Adjacency List
Exercise

What are the adjacency lists that represent this graph?

A: ?
B: ?
C: ?
D: ?
Adjacency list implementation #2

**Array**, where each element contains **linked list** of adjacent vertex labels

Requires: labels are integers; bounded number of vertices
Adjacency “matrix”

• Given integer labels and bounded # of vertices...

• Maintain a 2D Boolean array $b$

• Invariant: element $b[i][j]$ is true iff there is an edge from vertex $i$ to vertex $j$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Adjacency “matrix” with edge labels

- Invariant: Element $b[i][j]$ is $e$ iff there is an edge labeled $e$ from vertex $i$ to vertex $j$. Otherwise $b[i][j]$ is null.
Code Demo: Adjacency Matrix
Exercise

What is the adjacency matrix that represents this graph?

*(converting vertex labels to 0-based position in alphabet)*

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficiency of Representations

(if we have time)
Adjacency list  vs.  Adjacency matrix

Efficiency: Space to store

\[ O(|V| + |E|) \quad \text{vs.} \quad O(|V|^2) \]
# Adjacency list vs. Adjacency matrix

## Efficiency: Time to visit all edges?

| $|V|$ | $|E|$ | Adjacency list | Adjacency matrix |
|-----|-----|---------------|------------------|
| 4   | 10  | $O(|V| + |E|)$ | $O(|V|^2)$       |
Adjacency list vs. Adjacency matrix

Efficiency: Time to determine whether edge from $v_1$ to $v_2$ exists

$O(|V| + |E|)$  $O(1)$
Adjacency list vs. Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>
Adjacency list  vs.  Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>

Max # edges = $|V|^2$

Sparse

Dense
### Adjacency list vs. Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>

**Max # edges =** $|V|^2$

**Sparse:** $|E| \ll |V|^2$

**Dense:** $|E| \approx |V|^2$
## Adjacency list vs. Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>

**Sparse graphs** is better for **Dense graphs**.

- **Sparse graphs**: $|E| \ll |V|^2$
- **Dense graphs**: $|E| \approx |V|^2$

Max # edges = $|V|^2$
CS 2110
Lecture 13b/14

Graph traversals
• Breadth-first and depth-first search
• Topological sorting
# Adjacency list vs. Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>

How are $|V|$ and $|E|$ related?
## Limiting cases (graph sparsity)

<table>
<thead>
<tr>
<th>Points (disconnected)</th>
<th>Tree (connected)</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>• $</td>
<td>E</td>
<td>\approx 0$</td>
</tr>
<tr>
<td>• $O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>
Adjacency list vs. Adjacency matrix

<table>
<thead>
<tr>
<th>List</th>
<th>Property</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>

Sparse graphs are better for dense graphs.

Sparse: $|E| \ll |V|^2$

Dense: $|E| \approx |V|^2$

Max # edges = $|V|^2$
Hash Tables For Dummies
The **dictionary** ADT

- Given a word, return its definition
- Given a contact’s name, return their phone number
- Given a student’s ID number, return their NetID

- All involve looking up a **value** associated with some **key**

- Also known as:
  - Map
  - Associative array
Dictionary interface

Generic on two type parameters

```java
interface Map<K, V> {...}
```

- K: Type of keys
- V: Type of values

- Like an array, but with objects (instead of ints) as “indices”
- Keys are unique (a key can only map to one value)
  - But value could be a list

- `put(key: K, value: V)`
  Associate value with key
- `get(key : K): V`
  Return value associated with key
- `remove(key: K)`
  Remove any association for key
- `keySet(): Set<K>`
  Allow iterating over keys
- `containsKey()`, `size()`, ...
Example client code

Map<String, LocalDate> bdays = ...;

bdays.put("Alan Turing", LocalDate.of(1912, 6, 23));
bdays.put("John von Neumann", LocalDate.of(1903, 12, 28));

println("Turing was born on " + bdays.get("Alan Turing");
for (String name : bdays.keySet()) {
    println(name + " was born on " + bdays.get(name));
}
println("Do I know Katherine Johnson’s birthday? " +
    (bdays.containsKey("Katherine Johnson") ? "yes" : "no"));
Data structures for implementing a Dictionary

- Unsorted list
  - Put: O(1)
  - Get: O(N)

- Sorted list
  - Put: O(N)
  - Get: O(log N)
  - Keys must be comparable

- Binary search tree
  - Put: Ω(log N) – O(N)
  - Get: Ω(log N) – O(N)
  - Keys must be comparable

- Desired: array-like
  - Put: O(1)
  - Get: O(1)
  - Any key type
If keys were integers, could just use an array

- How hard could it be?
  - How to turn ordinary objects (keys) into integers?
  - How big an array will be needed?
  - What if two keys correspond to the same index?

- **Hash**: “to chop to pieces; to make a confused muddle of”
Example

- $h(\text{“Turing”}) \rightarrow 3$
- $h(\text{“von Neumann”}) \rightarrow 7$
- $h(\text{“Johnson”}) \rightarrow 5$

```java
class Entry<K, V> {
    K key;
    V value;
}
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>(Turing, 1912-06-23)</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>(Johnson, 1918-08-26)</td>
</tr>
<tr>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>7</td>
<td>(von Neumann, 1903-12-28)</td>
</tr>
</tbody>
</table>
Graph traversals
Traversal objectives

• Discover which vertices are reachable from a starting vertex
• Find a path from a starting vertex to a destination vertex
• “Visit” nodes in a particular order
  • Should only visit each node once, even if it is reachable via multiple paths

• More of a “pattern” than a specific algorithm
  • Outputs, “visit” procedure depend on task
Traversing applications

Find a public transit itinerary

Navigate maps in a game

Solve a maze

Images: https://new.mta.info/maps
Graph operations to support traversal

• Enumerate all nodes adjacent to a given node
  • Undirected: “neighbors”
  • Directed: “successors”

• Will assume adjacency list representation for analysis
  • Enumerating edges only considers edges that exist (instead of all possible edges in a row of a matrix)
  • Enumerating every edge leaving every node: $O(|E|)$ cost
Breadth-first search
Analogy: browsing with tabs
BFS Algorithm

Key idea: Iteratively visit each unvisited “layer”

Many possible BFS orders depending on how nodes in layer are chosen:
1, 2, 5, 7, 3, 8; 1, 7, 5, 2, 8, 3; 1, 5, 7, 2, 3, 8; (odds first)
BFS layers

• Layer indicates number of hops to get from *start* to any vertex in that layer

• Must discover all nodes in one layer before looking at any in the next layer
  • Discovered nodes can be put into a **FIFO queue** so that neighbors’ neighbors aren’t visited out of turn
  • Queue only contains vertices in current layer or next layer
BFS code

```java
Queue<Vertex> frontier = new LinkedList<>();
start.discovered = true;
frontier.add(start);
while (!frontier.isEmpty()) {
    Vertex v = frontier.remove();
    for (Vertex neighbor : v.successors()) {
        if (!neighbor.discovered) {
            neighbor.discovered = true;
            frontier.add(neighbor);
        }
    }
    // v is now “settled”
}
```

“frontier” is TODO list of vertices who might have undiscovered neighbors

Examine next vertex in queue until queue is empty

If vertex has any undiscovered neighbors, “discover” them and add them to the queue
BFS example

Frontier:

Discovered:

Layers:
BFS Order

Which of these is a BFS visit order? Assume lower-numbered neighbors are visited first.

A. 1, 2, 3, 4, 5, 6, 6, 7, 7, 8
B. 1, 2, 3, 4, 5, 6, 7, 8
C. 1, 2, 5, 3, 6, 4, 7, 8
D. 1, 2, 5, 6, 3, 7, 4, 8
Which of these is a BFS visit order? Assume lower-numbered neighbors are visited first.

A: 1, 2, 3, 4, 5, 6, 6, 7, 7, 8

B: 1, 2, 3, 4, 5, 6, 7, 8

C: 1, 2, 5, 3, 6, 4, 7, 8

D: 1, 2, 5, 6, 3, 7, 4, 8

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app
Time complexity of BFS

while (!frontier.isEmpty()) {
    Vertex v = frontier.remove();
    for (Vertex n : v.successors) {
        if (!n.discovered) {
            n.discovered = true;
            frontier.add(n);
        }
    }
}

|V| + 1  // True for each vertex, false once
|V|    // Each vertex removed once
|E|    // Consider every edge of every vertex
|E|
|V|    // Each vertex can only be discovered once
|V|

Total cost: $O(|V| + |E|)$
Space complexity of BFS

- Each vertex only added to frontier once
  - No worse than $O(|V|)$
- Worst-case: all vertices are neighbors of starting node
  - Could be as bad as $O(|V|)$

- Also need to track “discovered” status
  - $O(|V|)$

- Space complexity: $O(|V|)$
Depth-first search
Analogy: browsing without tabs, with back button
Analogy: navigating a maze
DFS algorithm

• Exhaustively search starting from one neighbor before moving on to next neighbor

• A search defined in terms of searches? This sounds like a job for…
  • Recursion!
DFS code

```java
void dfsVisit(Vertex start) {
    start.discovered = true;
    for (Vertex neighbor : start.successors()) {
        if (!neighbor.discovered) {
            dfsVisit(neighbor);
        }
    }
    // start is "settled"
}
```
DFS Order

Which of these is a DFS visit order? Assume lower-numbered neighbors are visited first.

A. 1, 2, 5, 6, 3, 6, 7, 4, 7, 8
B. 1, 2, 3, 4, 5, 6, 7, 8
C. 1, 2, 5, 3, 6, 4, 7, 8
D. 1, 2, 5, 6, 3, 7, 4, 8
Which of these is a DFS visit order? Assume lower-numbered neighbors are visited first.

A: 1, 2, 5, 6, 3, 6, 7, 4, 7, 8  
B: 1, 2, 3, 4, 5, 6, 7, 8  
C: 1, 2, 5, 3, 6, 4, 7, 8  
D: 1, 2, 5, 6, 3, 7, 4, 8
DFS efficiency

Time
- One recursive call per (reachable) vertex: $O(|V|)$
  - Each vertex can only be discovered once
- Must follow every edge once to check if end vertex is already discovered: $O(|E|)$
- Total: $O(|V|+|E|)$

Space
- One activation record per active recursive call
- Worst case: long chain
  - Depth of recursion: $O(|V|)$
- Tracking “discovered” status
  - $O(|V|)$
Iterative DFS

• Can replace recursion with loop by using a stack
  • For large graphs, the call stack might not be big enough

• Replacing BFS queue with a stack yields a DFS-like algorithm
  • But must allow duplicates in frontier to get correct order; hurts efficiency

• DSAJ uses better, but more complicated approach
  • Mimics the way recursion uses the call stack
  • Complexity hidden in pseudocode: “get unvisited neighbor” (preferably without looping)

• Upshot: DFS wants to be recursive!
Traversals tidbits

• Discovery edges form a tree
  • Starting vertex is root
• BFS easily computes shortest path length (not weight) from start to all other vertices
  • Discovery tree nodes are enumerated in level order
• With DFS, discovery order and settled order are different
  • Analogous to tree preorder vs. postorder
• If graph is not strongly connected, may not find all nodes from starting vertex
  • May need multiple traversals, starting from remaining vertices
Applications

• BFS-like algorithms are used to find shortest paths
  • Google Maps

• BFS visits nearby vertices sooner

• DFS models discovering paths by walking
  • Exploring a maze

• DFS can be used for topological sorting (dis11)
Topological sorting
Problem: find an order in which you can take courses without violating prerequisites

e.g. 1110, 2110, 2800, 3110, 3410, 4410, 4820
Topological order

A **topological order** of directed graph $G$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$, such that for every edge $(v_i, v_j)$, it holds that $i < j$.

**Intuition:** line up the nodes with all edges pointing left to right.

*Other applications:* robot planning, job scheduling, compilers
Cycles

- A directed graph can be topologically ordered if and only if it has no cycles.
- A **cycle** is a path $v_0, v_1, ..., v_p$ such that $v_0 = v_p$.
- A graph is **acyclic** if it has no cycles.
- A directed acyclic graph is a **DAG**.
Is this graph a DAG?

- If a node is part of a cycle, it must have an incoming edge
- Deleting a node with indegree zero would not remove any cycles
- Keep deleting such nodes and see whether graph “disappears”

Yes! It was a DAG.

The order in which we removed nodes was a topological order!
Topological sort: algorithm 1

While (there is a vertex v with no incoming edges):
    Append v to result
    Remove all of v’s outgoing edges from graph
If vertices with incoming edges remain, a cycle exists

• Don’t want to actually mutate graph, so instead:
  • Count in-degree of all vertices (store in dictionary)
  • Add all vertices with in-degree 0 to list
  • While list is not empty:
    • Remove a vertex and add to result
    • Subtract in-degree of all neighbors by 1