CompSci 201, L22: Graphs, DFS
Person in CS: Katherine Johnson

Katherine Johnson
Logistics, coming up

• Today, Wednesday, April 5
  • APT 8 due

• Next Monday, April 10
  • Project P5: Huffman due

• Next Wednesday, April 12
  • APT Quiz 2 due (will release on Saturday)
  • Covers linked list, sorting, trees
  • No regular APTs this week, just the quiz
What is a graph?

A **graph** is a data structure for representing connections among items, and consists of vertices connected by edges.

- A **vertex** (or node) represents an item in a graph.
- An **edge** represents a connection between two vertices in a graph.

Maps/directions software:
- Vertices \(\sim\) intersections
- Edges \(\sim\) roads

Zybook chapter 23
Undirected versus directed graphs

**Undirected Graph**
Edges go both ways

- Facebook network, most road networks, are undirected

**Directed Graph**
Edges go one way only

- Worldwide web is a directed graph of webpages (nodes) and links (directed edges)

Zybook chapter 23
Simple Graphs and Graph Sizes

• In a **simple** graph, there is at most one (undirected) edge between nodes (or 2 directed).

![Graph Diagram]

• Usually parameterize the size of the graph as:
  • $N$ (or $|V|$) = number of vertices/nodes
  • $M$ (or $|E|$) = number of edges
    • $M \leq N^2$ for a **simple** graph

Zybook chapter 23
Paths in Graphs

• A simple **path** is a sequence of unique vertices where subsequent nodes are connected by edges
  • (Also commonly defined as a sequence of edges with unique vertices)

• Example in bold blue: [A, B, F, G].
  • (or [e1, e4, e7])
Pathfinding or Graph Search

Is there a way to get from point A to point B?
- Maps/directions
- Video games
- Robot motion planning
- Etc.
Recursive Depth-first search (DFS) in Grid Graphs
Two-dimensional grid is simple graph with implicit structure

Represent as 2d array, e.g., char[][]
Two nodes adjacent if indices +/- 1

If not all of these edges are present, can represent a maze.
A maze is a grid graph

- Example: ten by ten grid
- Edge = no wall, no edge = wall.
- Look for a path from start (lower left) to middle.

```java
public class MazeDemo {
    private int mySize;
    private boolean[][] north; // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
    private boolean[][] west;
}
```
Depth First Search for Solving Maze

coursework.cs.duke.edu/cs-201-spring-23/maze-demo

Always explore (recurse on) a new (unvisited) adjacent vertex if possible.

If impossible, **backtrack** to the most recent vertex adjacent to an unvisited vertex and continue.
How is DFS Graph Traversal like Recursive Tree Traversal?

Tree traversals assumed only two adjacent nodes (children) and no cycles

• Just try recursing on every adjacent vertex?
• Unlike in a tree, there are cycles: How do we avoid infinite recursion?
Base Cases and Visited Set

23  private boolean[][] visited;

160  private int solveDFS(int x, int y, int depth) {
161     if (x == 0 || y == 0 || x == mySize + 1 || y == mySize + 1) return 0;
162     if (visited[x][y]) return 0;
163     visited[x][y] = true;

• Line 161: Base case: Searching off the grid
• Line 162: Base case: Already explored here

171     // reached middle which is goal of maze
172     if (x == mySize / 2 && y == mySize / 2) {
173         return depth;
174     }

• Line 172: Base case: Found the middle!
Recursive case

if (!north[x][y]) {
    int d = solveDFS(x, y + 1, depth+1);
    if (d > 0) return d;
}

3 more symmetric cases for other 3 directions
Runtime complexity for Recursive DFS maze/grid

• Suppose the grid has \( N = \text{width} \times \text{height} \) nodes.

• Each node will be recursed on \( \leq 4 \) times:
  • Has 4 neighbors that could recurse on it,
  • Keep track of visited, we don’t recurse from the same neighbor twice.

• Each recursive call is \( O(1) \).
• Overall runtime complexity is \( O(N) \).
Depth-First Search in General Graphs
General data structures for graphs: Not necessarily a grid

Adjacency List

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Adjacent vertices (edges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td>A, C, D</td>
</tr>
<tr>
<td>C</td>
<td>A, B</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Zybook chapter 23
Efficient Adjacency “List” Using Double Hashing

- HashMap<Vertex, HashSet<Vertex>> aList
  - Vertex type can be Integer, char, String, custom object, ..., needs to have good hashCode() and equals().

- aList.put('A', new HashSet())
- aList.get('A').add('B')
- aList.get('A').add('C')
- ...

\[ O(1) \text{ time to check if nodes are connected or get the neighbors of a node (assuming good hashCode)} \]
Graph Search Data Structures

• Have an adjacency list for the graph
• Keep track of visited nodes in a set
• Keep track of the previous node: During search, how did I get to this node?

Example has Character nodes, could be any label for the nodes.

Storing as instance variables, accessible in methods.

```java
9 public class DFS {
10   public static Map<Character, Set<Character>> aList;
11   public static Set<Character> visited;
12   public static Map<Character, Character> previous;
```
Recursive DFS on a General Graph: Visiting all nodes

```java
def dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```
Initialize search at A

Adjacency List:
- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```
Recurse on B

Visited (set)

{A, B}

Adjacency List:
A=\{B, D\}
B=\{A, E, F\}
C=\{F\}
D=\{A, E\}
E=\{B, D, F\}
F=\{B, C, E\}

```java
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```
Recurse on E

start: A

Adjacency List:
A=
[A, B, D]
B=
[A, E, F]
C=
[F]
D=
[A, E]
E=
[B, D, F]
F=
[B, C, E]

Visited (set)
{A, B, E}

```java
14   public static void dfs(char start) {
15       if (!visited.contains(start)) {
16           visited.add(start);
17           System.out.println(start);
18           for (char neighbor : aList.get(start)) {
19               dfs(neighbor);
20           }
21       }
22   }
```
Recurse on D

Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

start: A

Visited (set)
{A, B, E, D}

14    public static void dfs(char start) {
15        if (!visited.contains(start)) {
16            visited.add(start);
17            System.out.println(start);
18            for (char neighbor : aList.get(start)) {
19                dfs(neighbor);
20            }
21        }
22    }

11/14/22
Backtrack to E, recurse on F

Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

start: A

Visited (set)
{A, B, E, D, F}

```java
14   public static void dfs(char start) {
15       if (!visited.contains(start)) {
16           visited.add(start);
17           System.out.println(start);
18           for (char neighbor : aList.get(start)) {
19               dfs(neighbor);
20           }
21       }
22   }
```
Recurse on C

start: A

Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

Visited (set)
{A, B, E, D, F, C}

```java
14    public static void dfs(char start) {
15        if (!visited.contains(start)) {
16            visited.add(start);
17            System.out.println(start);
18            for (char neighbor : aList.get(start)) {
19                dfs(neighbor);
20            }
21        }
22    }
```
Did we really need recursion?  
preOrder Tree Traversal with Stack

```java
public static void preOrder(TreeNode tree) {
    Stack<TreeNode> myStack = new Stack<>();
    myStack.add(tree);
    while (!myStack.isEmpty()) {
        TreeNode current = myStack.pop();
        if (current != null) {
            System.out.println(current.info);
            myStack.add(current.right);
            myStack.add(current.left);
        }
    }
}
```

Recursion uses the call stack to keep track of nodes
Could also explicitly use a stack, can do the same for DFS
Stack Abstract Data Structure: LIFO List

public static void sdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Stack<String> st = new Stack<>();
    for(String s : strs) {
        st.push(s);
    }
    while (!st.isEmpty()) {
        System.out.println(st.pop());
    }
}

LIFO = Last In First Out

Push: Add element to stack

Pop: Get last element in

wonderful
is
compsci
Initializing Iterative DFS

- **Stack** stores nodes we have *visited/discovered*, but not explored from yet.
- Explore from one *current* node at a time.

```java
14     public static void dfs(char start) {
15         Stack<Character> toExplore = new Stack<>();
16         char current = start;
17         toExplore.add(current);
18         visited.add(current);
```

- Stack is LIFO (last-in first-out), so we always explore from the *last node we discovered*, **depth-first**!
Iterative DFS Loop

While there are nodes we have not explored from...

```java
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : alist.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

Explore from the most recently discovered node...

Look at all neighbors of current node...

If we haven’t seen them before...

Then:
1. note how we got here
2. Note we have seen
3. Mark to explore later
Initialize search at A

Adjacency List:
- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

start: A

- toExplore (stack)
  - A

- previous (map)
  - {A}

- Visited (set)
  - {A}
Pop A off the stack

start: A

Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack) previous (map) Visited (set)

{A}
Find B from A

Adjacency List:
- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

start: A

toExplore (stack) previous (map) Visited (set)
- B B <- A {A, B}
Find D from A

start: A

Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)   previous (map)   Visited (set)
D                B <- A          {A, B, D}
B
D <- A
Pop D off the stack

Adjugency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

start: A

toExplore (stack) previous (map) Visited (set)

B B <- A {A, B, D}
D D <- A
Find E from D

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

toExplore (stack)

E  B ← A
B  D ← A
D  E ← D

previous (map)

Visited (set)
{A, B, D, E}
Pop E off the stack

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

toExplore (stack)  previous (map)  Visited (set)
B  B <- A  {A, B, D, E}
D  D <- A
E  E <- D
Find F from E

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

start: A

toExplore (stack)  previous (map)  Visited (set)
F  B  <-  A  {A, B, D, E, F}
B
D  <-  A
E  <-  D
F  <-  E

Compsci 201, Fall 2022, L22: Graphs, DFS
Pop F off the stack

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

toExplore (stack) previous (map) Visited (set)

B
D
E
F
B <- A
D <- A
E <- D
F <- E

{A, B, D, E, F}
Find C from F

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

start: A

toExplore (stack)  previous (map)  Visited (set)

C  B <- A  {A, B, D, E, F, C}
B
D <- A
E <- D
F <- E
C <- F
Pop C off the stack

**Adjacency List:**
- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

**start:** A

<table>
<thead>
<tr>
<th>toExplore (stack)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B &lt;- A</td>
<td>{A, B, D, E, F, C}</td>
</tr>
<tr>
<td>D</td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E &lt;- D</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F &lt;- E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C &lt;- F</td>
<td></td>
</tr>
</tbody>
</table>
Pop B off the stack

Adjacency List:
A = [B, D]
B = [A, E, F]
C = [F]
D = [A, E]
E = [B, D, F]
F = [B, C, E]

toExplore (stack) | previous (map) | Visited (set)
B <- A
D <- A
E <- D
F <- E
C <- F

{A, B, D, E, F, C}
DFS Search Tree

Adjacency List:
- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

start: A

toExplore (stack)   previous (map)   Visited (set)
B <- A               E <- D          {A, B, D, E, F, C}
D <- A               F <- E
E <- D               C <- F

Path from A to C:  
C <- F <- E <- D <- A

Can find paths from A to X by following previous backwards from X