CompSci 201, L23: Graphs, BFS
Logistics, coming up

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

• This Wednesday, April 12
  • APT Quiz 2 due
  • Covers linked list, sorting, trees
  • No regular APTs this week, just the quiz
Today’s agenda

• Wrap up depth-first search (DFS)

• Introduce breadth-first search (BFS)

• Example WordLadder Problem
Pathfinding or Graph Search

Is there a way to get from point A to point B?

- Maps/directions
- Video games
- Robot motion planning
- Etc.
General data structures for graphs

Adjacency List

![Adjacency List Diagram]

<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>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</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>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Zybook chapter 23
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?

```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;
```

• Example has Character nodes, could be any label for the nodes.
• Storing as instance variables, accessible in methods.
DFS Complexity?

```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);
        }
    }
}
```

While loop over all nodes (N), potentially?
Loop over edges (M)

Seems like $O(NM)$, but...
DFS Complexity?

```
20 while (!toExplore.isEmpty()) {
21     current = toExplore.pop();
22     for (char neighbor : aList.get(current)) {
23         if (!visited.contains(neighbor)) {
24             previous.put(neighbor, current);
25             visited.add(neighbor);
26             toExplore.push(neighbor);
27         }
28     }
29 }
```

- Pop each of N nodes at most once.
- Loop over neighbors of each node exactly once, considers each edge twice.
- N+2M is O(N+M).
Iterative Breadth-First Search (BFS)
Queue: A FIFO List

- Both add and remove are $O(1)$
  - Add at end of LinkedList
  - Remove from front of LinkedList

```java
public static void qdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Queue<String> q = new LinkedList<>();
    for (String s : strs) {
        q.add(s);
    }
    while (!q.isEmpty()) {
        System.out.println(q.remove());
    }
}
```
levelOrder Tree Traversal with a queue

public static void levelOrder(TreeNode tree) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(tree);
    while (!queue.isEmpty()) {
        TreeNode current = queue.remove();
        if (current != null) {
            System.out.println(current.info);
            queue.add(current.left);
            queue.add(current.right);
        }
    }
}

Use a queue to keep track of nodes
First in first out, nodes visited in level order
Depth First Search for Solving Maze

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.

[coursework.cs.duke.edu/cs-201-fall-22/maze-demo](coursework.cs.duke.edu/cs-201-fall-22/maze-demo)
Breadth First Search for Solving Maze

Explore *all* your neighbors (adjacent vertices) before you visit any of your neighbors’ neighbors.

Looking for the shortest path/solution.

coursework.cs.duke.edu/cs-201-fall-22/maze-demo
Queue = BFS, Stack = DFS

BFS: FIFO Exploration
search all locations one-away from start, then two-away, ...

DFS: LIFO Exploration
Search path as far as possible, backtrack if need to another branch...
Initializing Iterative BFS

• **Queue** stores nodes we have *visited/discovered*, but not explored from yet.

• Explore from one *current* node at a time.

```java
32  public static void bfs(char start) {
33      Queue<Character> toExplore = new LinkedList<>();
34      char current = start;
35      visited.add(current);
36      toExplore.add(current);
```

• Queue is FIFI(first-in first-out), so we always explore from the *first/closest (unvisited) node we discovered*, **breadth-first**!
Iterative BFS Loop

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

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

Explore from the closest 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]

toExplore (queue)  previous (map)  Visited (set)
A                     {A}
Remove A from the queue

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

**toExplore (queue)**   **previous (map)**   **Visited (set)**

{A}
Find B 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 (queue)  previous (map)  Visited (set)
B  B <- A  {A, B}
Find D 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 (queue)  previous (map)  Visited (set)
B  B <- A  {A, B, D}
D  D <- A

Note the difference, add to end of queue!
Remove B from queue

Adjacency List:

- A = [B, D]
- B = [A, E, F]
- C = [F]
- D = [A, E]
- E = [B, D, F]
- F = [B, C, E]

start: A

B was first in, B is first out

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>B &lt;- A</td>
<td>{A, B, D}</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4/10/23
Compsci 201, Spring 2023, L23: Graphs BFS
Find E from B

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

**toExplore (queue)**

<table>
<thead>
<tr>
<th>vertex</th>
<th>previous (map)</th>
<th>visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>B &lt;- A</td>
<td>{A, B, D, E}</td>
</tr>
<tr>
<td>E</td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E &lt;- B</td>
<td></td>
</tr>
</tbody>
</table>
Find F from B

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

toExplore (queue)  previous (map)  Visited (set)

D  B ← A  \{A, B, D, E, F\}
E  D ← A
F  E ← B
     F ← B
Remove D from queue

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 (queue)  previous (map)  Visited (set)
E  B <- A  \{A, B, D, E, F\}
F  D <- A
   E <- B
   F <- B
Remove E from queue

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 (queue)  previous (map)  Visited (set)
F          B <- A          {A, B, D, E, F}
D          A
E          B
F          B
Remove F from queue

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

{A, B, D, E, F, C}
Remove C from queue

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 (queue) previous (map) Visited (set)

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

{A, B, D, E, F, C}
BFS 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 (queue)  previous (map)  Visited (set)
B ← A
D ← A
E ← B
F ← B
C ← F

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

Comparing DFS and BFS Search
Trees

start: A

previous (map)

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

Length 4 path from A to C

start: A

previous (map)

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

Length 3 path from A to C, shorter!
Pathfinding Properties

• DFS and BFS **both** find valid paths to _all_ nodes reachable from the start.
  • Can return early if you only want to find a path to a specific target node

• BFS finds the _shortest path_ to every reachable node, DFS does _not_ guarantee this.
Example WordLadder Problem

A **transformation sequence** from word \textit{beginWord} to word \textit{endWord} using a dictionary \textit{wordList} is a sequence of words \texttt{beginWord} \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every \( s_i \) for \( 1 \leq i \leq k \) is in \textit{wordList}. Note that \texttt{beginWord} does not need to be in \textit{wordList}.
- \( s_k = \text{endWord} \)

Given two words, \texttt{beginWord} and \texttt{endWord}, and a dictionary \textit{wordList}, return the **number of words in the shortest transformation sequence** from \texttt{beginWord} to \texttt{endWord}, or 0 if no such sequence exists.

[leetcode.com/problems/word-ladder/description/](https://leetcode.com/problems/word-ladder/description/)