CompSci 201, L24: Shortest Paths in Weighted Graphs
Person in CS: Edsger Dijkstra

• Dutch computer scientist, 1930 – 2002.
• PhD in 1952, Turing award in 1972.
• Originally planned to study law, switched to physics, then to computer science.
• “After having programmed for some three years...I had to make up my mind, either to...become a...theoretical physicist, or to ...become..... what? A programmer? But was that a respectable profession?....Full of misgivings I knocked on Van Wijngaarden's office door, asking him whether I could "speak to him for a moment"; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently...he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come?”
Logistics, Coming up

• APT Quiz 2 due today tomorrow, Tuesday 11/22
  • covers APT6-10, linked list and tree problems guaranteed
  • Start by 10 pm to get full 2 hours

• Midterm Exam 3 on Wednesday 11/30 Monday 12/5

• Project 6: Available, due Monday 12/5 Wednesday 12/7
Midterm Exam 3

• Logistics:
  • 60 minutes, in-person, short answer
  • Can bring 1 reference/notes page

• Topics:
  • Trees, binary search trees, binary heaps, recursion
  • Red-black trees: Properties and implications, yes, details of rebalance algorithm, no.
  • Greedy, Huffman
  • Graphs, DFS, BFS, Dijkstra’s

• Practice exam will release week we get back from break
Project 6: Route Demo

Partner project, can work (and submit) with one other person. Make sure to read the directions on using Git with a partner, and to submit together on gradescope.

Two parts:

1. GraphProcessor: Implement algorithms with real-world graph data, and
2. GraphDemo: Make and record a demo. Example minimal demo here.

No analysis for this project.
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...

```java
38     while (!toExplore.isEmpty()) {
39         current = toExplore.remove();
40         for (char neighbor : aList.get(current)) {
41             if (!visited.contains(neighbor)) {
42                 previous.put(neighbor, current);
43                 visited.add(neighbor);
44                 toExplore.add(neighbor);
45             }
46         }
47     }
```

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
Weighted Graphs and Dijkstra’s Algorithm
Weighted Graphs

Each edge has an associated \textit{weight} representing cost, distance, etc.

In mapping applications, maybe one road is twice as long as another.
Project 6: Route

Durham, NC → Seattle WA,
~2800 miles
Shortest weighted paths?

• BFS gives shortest paths in *unweighted* graphs.

• Modify BFS to account for weights; called Dijkstra’s algorithm.

• BFS = queue, Dijkstra’s = ...
  • Priority queue!
Exploring a node with Dijkstra’s Algorithm, Pseudocode

While unexplored nodes remain
• Explore current = the closest unexplored node
• For each neighbor:
  • Update shortest path to neighbor if shorter to go through current

Just like BFS (explore closer nodes first) except...now we need to account for weights.
Initializing Dijkstra’s Algorithm

Ordering priority by *distance of the shortest path found so far*, to a given node.

```java
public static Map<Character, Integer> dijkstra(char start) {
    Map<Character, Integer> distance = new HashMap<>();
    Comparator<Character> comp = (a, b) ->
        distance.get(a) - distance.get(b);
    PriorityQueue<Character> toExplore = new PriorityQueue<>(comp);

    char current = start;
    distance.put(current, value: 0);
    toExplore.add(current);
```

Going to choose to explore the *next closest unexplored node to start* at each iteration.
The search loop in general

Keep searching while there are unexplored nodes.

Choose to explore from the next closest (to start) unexplored node to start at each iteration.

```java
66 while (!toExplore.isEmpty()) {
67     current = toExplore.remove();
68     for (char neighbor : alist.get(current)) {...
77 }
```

Search all neighbors of current. If you find a shorter path to neighbor through current, update to reflect that.
Details: Checking each neighbor

Weight/distance on current -> neighbor edge.

```
for (char neighbor : aList.get(current)) {
    int weight = getWeight(current, neighbor);
    if (!distance.containsKey(neighbor)
        && distance.get(neighbor) > distance.get(current) + weight) {
        distance.put(neighbor, distance.get(current) + weight);
        previous.put(neighbor, current);
        toExplore.add(neighbor);
    }
}
```

Just like BFS (explore closer nodes first) except...now we need to account for weights.

If no path found yet...

Or we found a shorter path to neighbor by going through current...

Record the new shortest path to neighbor, and add neighbor to explore later.
Duplicates in the PriorityQueue

• Note that we might add the same node to the PriorityQueue multiple times 😞

```java
71       if (!distance.containsKey(neighbor))
72           // distance.get(neighbor) > distance.get(current) + weight) {
73               distance.put(neighbor, distance.get(current) + weight);
74               previous.put(neighbor, current);
75               toExplore.add(neighbor);
76           }
```

• In textbooks, line 76 usually *updates the priority* of neighbor, not add to the PriorityQueue.

• But most standard library binary heaps (including java.util) don’t support an efficient update priority operation. So we add again with the new priority.
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 (PriorityQueue)  previous (map)  distance (map)
A
A = 0
Remove A from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
A = 0
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]

toExplore (PriorityQueue)  previous (map)  distance (map)
B

B <- A
A = 0
B = 2 (A + 2)
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]

toExplore (PriorityQueue)
D
B

previous (map)
B <- A
D <- A

distance (map)
A = 0
B = 2
D = 1 (A + 1)
Remove D from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
B  B <- A  A = 0
D <- A  B = 2
  D = 1
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 (PriorityQueue)  previous (map)  distance (map)
B  B <- A  A = 0
E  D <- A  B = 2
    E <- D  D = 1
      E = 2 (D + 1)

B and E are tied in distance, suppose B comes first
Remove B from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
E          B <- A
           D <- A
           E <- D

A = 0
B = 2
D = 1
E = 2
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 (PriorityQueue)

previous (map)
B <- A
D <- A
E <- D
F <- B

distance (map)
A = 0
B = 2
D = 1
E = 2
F = 5 (B + 3)

E has lower distance
Remove E from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
F  B <- A  A = 0
    D <- A  B = 2
    E <- D  D = 1
    F <- B  E = 2
               F = 5
Find **shorter** path to 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]

<table>
<thead>
<tr>
<th>toExplore (PriorityQueue)</th>
<th>previous (map)</th>
<th>distance (map)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F [new dist of 4]</td>
<td>B &lt;- A</td>
<td>A = 0</td>
</tr>
<tr>
<td></td>
<td>E &lt;- D</td>
<td>D = 1</td>
</tr>
<tr>
<td></td>
<td>F &lt;- E</td>
<td>E = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F = 4 (E + 2)</td>
</tr>
</tbody>
</table>

*Hack because java.util.PriorityQueue cannot decrease key*
Remove F from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
F [old dist of 5]  B <- A  A = 0
                  D <- A  B = 2
                  E <- D  D = 1
                  F <- E  E = 2

Hack because java.util.PriorityQueue cannot decrease key dist
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]

toExplore (PriorityQueue)  previous (map)  distance (map)
F [old dist of 5]  B <- A  A = 0
C                    D <- A  B = 2
                      E <- D  D = 1
                      F <- E  E = 2
                      C <- F  F = 4
                      C = 5 (F + 1)
Remove old F from PriorityQueue

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

toExplore (PriorityQueue)  previous (map)  distance (map)
C
B <- A
D <- A
E <- D
F <- E
C <- F

A = 0
B = 2
D = 1
E = 2
F = 4
C = 5
Remove C from PriorityQueue

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

toExplore (PriorityQueue) | previous (map) | distance (map)
--- | --- | ---
B <- A | A = 0 |
D <- A | B = 2 |
E <- D | D = 1 |
F <- E | E = 2 |
C <- F | F = 4 |

11/21/22 Compsci 201, Fall 2022, L24: Shortest Paths 31
Go to duke.is/b3v2w

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
Is Dijkstra’s algorithm guaranteed to be correct? (Informal)

• **Claim.** Distance is correct shortest path distance for all nodes *explored* so far, and shortest path distance *through explored nodes* for all others.

• Formal proof is *by induction*, see Compsci 230.
  • Assume the property is true up to some point in the algorithm, then...
  • Consider the next node we explore:
Is Dijkstra’s algorithm guaranteed to be correct? (Informal)

The shortest path distance so far goes through explored nodes.

Suppose we explore from C this iteration.

Can’t be another shorter path through an unexplored node, there would be a node that should have been removed first instead of C.

Explored nodes

W(A, C)
d[A]
d[B]
Runtime Complexity of Dijkstra’s Algorithm (with N nodes, M edges)

Like BFS, consider each node once and each edge twice, \( \log(N) \) operations for each:

\[ O((N+M)\log(N)) \]
Problem with Heap Duplicates

- In graphs with constant degree (where each node has at most a constant number of neighbors), will still just be O(N) iterations, maybe not N.
- For general graphs worst-case provable O((N+M) log(N)) need an efficient priority update.

```java
66 while (!toExplore.isEmpty()) {
67     current = toExplore.remove();
68     for (char neighbor : aList.get(current)) {...
77 }
```

May actually loop more than N times
Reviewing Recursion, Greedy
Optimal Apple Collecting Problem

https://leetcode.com/problems/minimum-time-to-collect-all-apples-in-a-tree/

Optimization with DFS

Given an undirected tree consisting of n vertices numbered from 0 to n-1, which has some apples in the vertices. You spend 1 second to walk over one edge of the tree. Return the minimum time in seconds you have to spend in order to collect all apples in the tree starting at vertex 0 and coming back to this vertex.

\[ n = 7, \text{ edges } = \{[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]\} \]
\[ \text{hasApple} = [\text{false},\text{false},\text{true},\text{false},\text{true},\text{true},\text{true},\text{false}] \]

return: 8
Directed -> Undirected Tree

• What is a tree?
  • We have explored binary (search) trees
  • With parent -> child, child does not point to parent
  • Directed

• Graph theory: what is a tree?
  • connected, acyclic, undirected graph
Thinking recursively

Time (# edges to traverse) to collect all apples at and below me is...

- Node 0: 0, nothing to do
- Node 2: 0, nothing to do, I have an apple, but my children don’t.
- Node 4: 4, I have 2 children with apples & it takes 2 edges to get each.
- Node 5: 0, nothing to do
- Node 6: 0, nothing to do
- Node 8, takes 2 edges to get to my right child with an apple and 2 edges to get to my left child, who used 4 edges to collect apples at/below.
Another Example

Time (# edges to traverse) to collect all apples at and below me is...

2, I have a child with an apple & takes 2 edges to get there and back

6, 2 to get my right child’s apple and 2 to get to my left child who used 2 edges to collect apples at/below.

0, I have an apple, but my children don’t.
Generalizing an algorithm in pseudocode

\[ m_T = \text{time (\# edges to traverse) to collect all apples at and below me...} \]

- **Base case.** If leaf, return 0
- **Recursive case.** For each child \( c \):
  - Get \( c_T \), the time to collect apples at/below child
  - If \( c_T \) is 0 \& child has apple, add 2 to \( m_T \)
  - If \( c_T > 0 \), add \( c_T + 2 \) to \( m_T \)
  - Return \( m_T \)
Optimal Apple Collecting Problem

https://leetcode.com/problems/minimum-time-to-collect-all-apples-in-a-tree/

Live Coding (Time permitting)

Given an undirected tree consisting of n vertices numbered from 0 to n-1, which has some apples in the vertices. You spend 1 second to walk over one edge of the tree. Return the minimum time in seconds you have to spend in order to collect all apples in the tree starting at vertex 0 and coming back to this vertex.

n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]]
hasApple = [false,false,true,false,true,true,false]

return: 8