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

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

• Next Wednesday, 4/19
  • Midterm exam 3
  • APT 9 extended to Thursday 4/20
Midterm Exam 3

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

• Topics could include:
  • 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 exams will release by the weekend
Today’s agenda

• Finish Example WordLadder Problem

• Shortest paths in weighted graphs: Dijkstra’s algorithm
Example WordLadder Problem

A **transformation sequence** from word *beginWord* to word *endWord* using a dictionary *wordList* is a sequence of words *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 *wordList*. Note that *beginWord* does not need to be in *wordList*.
- $s_k = \text{endWord}$

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

[leetcod]e.com/problems/word-ladder/description/
Weighted Graphs and Dijkstra’s Algorithm
Weighted Graphs

Each edge has an associated 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
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.
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.
“Textbook” Dijkstra Initialization

• Initialize distances to:
  • 0 for the start node,
  • Infinity for everything else

• Add all nodes to a priority queue, using their distance as the priority.
“Textbook” Dijkstra Exploration

• While there are unexplored nodes:
  • Get the closest unexplored node to the start
  • Look at all neighbors:
    • If the path through current is shorter:
      • Update distance, update priority in priority

```java
12     while (toExplore.size() > 0) {
13         char current = toExplore.remove();
14         for (char neighbor : alist.get(current)) {
15             int newDist = distance.get(current) + getWeight(current, neighbor);
16             if (newDist < distance.get(neighbor)) {
17                 distance.put(neighbor, newDist);
18                 //toExplore.decreasePriority(neighbor);
19             }
20         }
21     }
22     return distance;
```
Practical Dijkstra Initialization

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

```java
public Map<Character, Integer> stdDijkstra(char start, Map<Character, List<Character>> aList) {
    Map<Character, Integer> distance = new HashMap<>();
    distance.put(start, 0);
    Comparator<Character> comp = (a, b) -> distance.get(a) - distance.get(b);
    PriorityQueue<Character> toExplore = new PriorityQueue<>(comp);
    toExplore.add(start);
}
```

Don’t need to add anything for all nodes yet
Practical Dijkstra search loop

Keep searching while there are unexplored nodes.

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

```java
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    for (char neighbor : aList.get(current)) {
        return distance;
    }

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

All neighbors of current node

Distance to neighbor through current = distance to current + weight on edge from current to neighbor

```java
for (char neighbor : aList.get(current)) {
    int newDist = distance.get(current) + getWeight(current, neighbor);
    if (!distance.containsKey(neighbor) || newDist < distance.get(neighbor)) {
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
}
```

If neighbor newly discovered OR found a shorter path...
- Record new distance
- Add to priority queue
Duplicates in the PriorityQueue

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

```java
for (char neighbor : aList.get(current)) {
    int newDist = distance.get(current) + getWeight(current, neighbor);
    if (!distance.containsKey(neighbor) || newDist < distance.get(neighbor)) {
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
}
```

• 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]

To explore (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                               D <- A    B = 2

4/12/23  Compsci 201, Spring 2023, L24: Shortest Paths  24
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)
- B
- E

previous (map)
- B ← A
- D ← A
- E ← D

distance (map)
- A = 0
- B = 2
- 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  A = 0
D <- A  B = 2
E <- D  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)  distance (map)
E  B ← A  A = 0
F  D ← A  B = 2

E has lower distance  D ← A  D = 1
E ← D  E = 2
F ← B  F = 5 (B + 3)
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]

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

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]

**toExplore** (PriorityQueue)
- F [new dist of 4]
- F [old dist of 5]

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

**distance** (map)
- A = 0
- B = 2
- D = 1
- E = 2
- F = 4 (E + 2)

*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
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)
F [old dist of 5]
C

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

distance (map)
A = 0
B = 2
D = 1
E = 2
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  A = 0
    D <- A  B = 2
    E <- D  D = 1
    F <- E  E = 2
    C <- F  F = 4
    C = 5
Remove C from PriorityQueue

Adjacency List:
\[
\begin{align*}
A &= [B, D] \\
B &= [A, E, F] \\
C &= [F] \\
D &= [A, E] \\
E &= [B, D, F] \\
F &= [B, C, E]
\end{align*}
\]

toExplore (PriorityQueue) | previous (map) | distance (map)
---|---|---
B <- A | A = 0 | B = 2
D <- A | B = 2 | D = 1
E <- D | D = 1 | E = 2
F <- E | E = 2 | F = 4
C <- F | F = 4 | C = 5
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.

The shortest path distance so far goes through explored nodes.

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

May actually loop more than N times

```java
33    while (toExplore.size() > 0) {
34        char current = toExplore.remove();
35        for (char neighbor : aList.get(current)) {
36            } // missing closing brace
37    return distance;
38```

• 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 queue update.