L2: Shortest Paths in Weighted Graphs

Alex Steiger
CompSci 201: Spring 2024
4/10/2024

Person in CS: Edsger Dijkstra

- 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 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 with him. After a long discussion, when I left the 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 be one of the persons called to make programming a respectable discipline in the years to come?"

Logistics, coming up

- Today, Wednesday, April 10
  - APT Quiz 2 due
  - Covers linked list and trees
  - No regular APTs due this week, just the quiz

- Next Wednesday, 4/17
  - Midterm exam 3
  - Practice exams coming soon to Canvas
  - APT 9 extended to Thursday 4/20
Midterm Exam 3

• Logistics:
  • 60 minutes, in-person, multiple-choice + fill-in-blank
  • Can bring 1 reference/notes page

• Topics could include:
  • Trees, binary search trees, binary heaps, recursion
  • AVL trees: High-level concept of balance factor/rotations, yes, details of performing rotations, no.
  • Greedy, Huffman
  • Graphs, DFS, BFS, Dijkstra’s

Today’s agenda

• Finish 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 -> s_1 -> s_2 -> ... -> s_k` such that:

• Every adjacent pair of words differs by a single letter.
• Every `s_i` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
• `s_k` = `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.

leetcode.com/problems/word-ladder/description/
Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

Suppose you have:

beginWord = "cat"
endWord = "dog"
wordList = ["hot","dot","dog","lot","log","cog","cot"]

The length of the shortest word ladder is... *

Consider this makeGraph method, part of a correct solution to the wordLadder problem. Assume the oneOff method correctly returns true if two strings differ by a single character and false otherwise, and runs in $O(1)$ time.

If $N$ is the length of the wordList, what is the asymptotic runtime complexity of the makeGraph method as a function of $N$? *

```java
23    private Map<String, HashSet<String>> makeGraph(List<String> wordList) {
24        Map<String, HashSet<String>> aList = new HashMap<>();
25        for (String w: wordList) {
26            aList.put(w, new HashSet<>());
27            for (String other: wordList) {
28                if (oneOff(w, other)) {
29                    aList.get(w).add(other);
30                }
31            }
32        }
33        return aList;
34    }
```
Consider this code, part of a correct solution to the wordLadder problem. It works with an adjacency list representation aList such as would be generated by the makeGraph method.

If there are \( N \) words in total in the wordList, and each word can be transformed into at most a constant number \( O(1) \) other words by changing a single character, then what is the runtime complexity of this code? *
```java
Queue<String> toExplore = new LinkedList<>();
Map<String, Integer> ladderLength = new HashMap<>();
toExplore.add(beginWord); ladderLength.put(beginWord, 1);

while (toExplore.size() > 0) {
    String word = toExplore.remove();
    for (String other : alist.get(word)) {
        if (!ladderLength.containsKey(other)) {
            ladderLength.put(other, ladderLength.get(word)+1);
            toExplore.add(other);
        }
    }
}
return ladderLength.getOrElse(endWord, 0);
```
For the approach to the wordLadder problem outlined above, what dominates the runtime complexity of the algorithm? Assume that each word is at most a constant length and that each word can be transformed into at most a constant number of other words.

- Building the graph takes most of the time
- Running the search on the graph takes most of the time
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
Google Maps Directions
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.

GraphProcessor: Implement algorithms with real-world graph data.

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

```java
public Map<Character, Integer> textbookDijkstra(Node start, Map<Character, List<Edge>> list) {
    // Implementation
}
```

“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 queue

```java
while (!unexplored.isEmpty()) {
    char current = unexplored.poll();
    for (char neighbor : graph.get(current)) {
        int newDist = distance.get(current) + graph.get(current, neighbor);
        if (newDist < distance.get(neighbor)) {
            distance.put(neighbor, newDist);
            // Update priority in priority queue
        }
    }
}
```

Practical Dijkstra Initialization

Like the previous implementation, but only add vertices to the queue once they are actually reached/visited.

```java
public Map<Character, Integer> textbookDijkstra(Node start, Map<Character, List<Edge>> list) {
    // Implementation
}
```

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

Keep searching while there are unexplored nodes.

```
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    int currDist = distance.get(current);
    for (char neighbor : allList.get(current)) {
        int newDist = currDist + weight on edge from current to neighbor;
        if (distance.containsKey(neighbor)) {
            if (newDist < distance.get(neighbor)) {
                distance.put(neighbor, newDist);
                toExplore.add(neighbor);
            }
        } else if (newDist < distance.get(neighbor)) {
            // implement decreasePriority by removal and re-insertion
            toExplore.remove(neighbor);
            distance.put(neighbor, newDist);
            toExplore.add(neighbor);
        }
    }
}
return distance;
```

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

Details: Checking each neighbor

All neighbors of current node

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

```
for (char neighbor : allList.get(current)) {
    int newDist = currDist + weight on edge from current to neighbor;
    if (distance.containsKey(neighbor)) {
        if (newDist < distance.get(neighbor)) {
            toExplore.add(neighbor);
        }
    } else if (newDist < distance.get(neighbor)) {
        // implement decreasePriority by removal and re-insertion
        toExplore.remove(neighbor);
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
}
```

Implementing decreasePriority

- Most standard library binary heaps (including java.util) don't support an efficient update/decrease priority operation.

```
else if (newDist < distance.get(neighbor)) {
    // implement decreasePriority by removal and re-insertion
    toExplore.remove(neighbor);
    distance.put(neighbor, newDist);
    toExplore.add(neighbor);
}
```

- Our code works, but is O(N) time
  - Java's PQ takes O(N) to remove given node (O(1) for smallest)
  - Other PQ implementations support O(log N)-time decreasePriority, but they are not in Java library
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

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

\[ B = 2(A + 2) \]
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) previous (map) distance (map)
D <- A D = A 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 <- A 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 (PriorityQueue) previous (map) distance (map)
B <- A D <- A E <- D

B and E are tied in distance, suppose B comes first
A = 0 B = 2 D = 1 E = 2 (D + 1)
Remove B from PriorityQueue

Find longer path to E from B

Find F from B
Remove E from PriorityQueue

Find shorter path to F from E

Remove F from PriorityQueue
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 | 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 | 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:
- 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
 |  | 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:

Runtime Complexity of Dijkstra's Algorithm (with N nodes, M edges)

**assuming O(log N) decreasePriority**

Like BFS, consider each node once and each edge twice, log(N) operations for each: $O((N+M)\log(N))$.