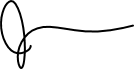
CS404, Fall Semester 2016: Find the Sneaky Path

I understand and have adhered to the rules regarding student conduct. All material, including algorithms and programs, have been produced and written by myself. Any outside sources that I have consulted are free, publicly available, and have been appropriately cited. I understand that a violation of the code of conduct will result in a zero (0) for this assignment, and that the situation will be discussed and forwarded to the Academic Dean of the School for any follow up action. It could result in being expelled from the university.

Justin Orel



11/27/16

**Introduction**

The Sneaky Path problem is a simple one. In a world where everyone exclusively chooses their path based on shortest distance, I want to find a path based on fewest fellow travelers, aka the Sneaky Path.

For this particular problem, two matrices are given: one which has the weights (distances) from each node (city, destination, etc) to every other node it has direct access to. This allows me to perform a Shortest-Path Algorithm (SPA) in order to determine the shortest path from every node to every other node. The second matrix contains the flow between all node pairings. This represents the number of people that want to get from one place to the next.

The basis of the problem is to find a way to synthesize the two matrices in order to find out how many people are actually travelling from each place to every other place. By combining the shortest paths with the flow matrix, it is possible to tell how many people will be on any given edge (road). Using the proper algorithms, we can find the path that allows us to reach our destination and do so while seeing the least number of other travelers.

**Algorithm Design and Analysis**

**Overview:**

First, I take the Edge Matrix that contains the weights between all connected nodes. Running this matrix through the Floyd-Warshall Shortest-Path Algorithm (discussed in greater detail later) creates two new matrices: the All-Pairs Shortest-Paths matrix and a Path Reconstruction matrix. The All-Pairs matrix will show the shortest distance between each pair of nodes. The Path Reconstruction matrix allows us to reconstruct the actual shortest path by displaying which node was traversed to get to the current node. The All-Pairs matrix allows us to know the actual distance between two nodes, but without a tool to track the traversed nodes, we wouldn’t know how to get there. It would be like knowing that the distance between Kansas City and Los Angeles is 1,576 miles, but not knowing what roads you would need to take to get you there.

Using the Path Reconstruction matrix, I work backwards through every node pairing to come up with the actual shortest paths between every combination of nodes. Once I have obtained all of the shortest paths, I can synthesize that with the Flow matrix in order to get the actual traffic on all roads. This is done by iterating through every path, and at each step of the path adding the flow between origin and destination to each leg of the path on a new Load Matrix. For example, let the origin be node 1 and the destination be node 4. Say the shortest path from 1 to 4 goes through nodes 5 and then 3 and the flow from 1 to 4 is 10 cars. In this case, 10 needs to be added to the Load matrix at points 1-5, 5-3, and 3-4. This process is repeated for every path.

Now that the Load (or traffic) Matrix is built, it is run through the Floyd-Warshall algorithm to produce a new All-Pairs matrix and a new Path Reconstruction matrix. The shortest paths portion from above is repeated in order to create shortest paths that represent the minimum traffic instead of the minimum distances. With the shortest, or Sneaky, paths calculated, can simply look up the sneakiest path between any combination of nodes.

Before going on to the technical details of the algorithm, more discussion of the Floyd-Warshall SPA is warranted, especially since it is central to my comprehensive algorithm. The following is from the Department of Mathematics at the Technical University of Munich,

“The Floyd-Warshall algorithm relies on the principle of dynamic pogramming. This means that all possible paths between pairs of nodes are being compared step by step, while only saving the best values found so far.

The algorithm begins with the following observation: If the shortest path from **u** to **v** passes through **w**, then the partial paths from u to w and w to v must be minimal as well. Correctness of this statement can be shown by induction. The algorithm of Floyd-Warshall works in an iterative way.

Let G be a graph with numbered vertices **1** to **N**. In the **k**th step, let **shortestPath(i,j,k)** be a function that yields the shortest path from **i** to **j**that only uses nodes from the set **{1, 2, ..., k}**. In the next step, the algorithm will then have to find the shortest paths between all pairs **(i, j)** using only the vertices from **{1, 2, ..., k, k + 1}**.

For all pairs of vertices, it holds that the shortest path must either only contain vertices in the set **{1, ..., k}**, or otherwise must be a path that goes from **i** to **j** via **k + 1**. This implies that in the **(k+1)**th step, the shortest path from **i** to **j** either remains **shortestPath(i,j,k)** or is being improved to **shortestPath(i,k+1,k) + shortestPath(k+1, j, k)**, depending on which of these paths is shorter. Therefore, each shortest path remains the same, or contains the node **k + 1** whenever it is improved.

This is the idea of dynamicprogramming. In each iteration, all pairs of nodes are assigned the cost for the shortest path found so far.”

**Pseudocode and Time Complexities:**

\**FloydWarshall and getShortestPath algorithms adapted from Wikipedia:*

<https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm>

**Algorithm** *FloydWarshall* (int[][] edge, int[][] pr):

*Input:* Integer martrix, edge, representing the weights between all node pairs

Integer matrix, pr, initialized to represent every node coming from itself

*Output:* Integer matrix, All-Pairs, with shortest distances between all nodes

Integer matrix, pr, with all values representing prior nodes in path

*N* 🡨edge.length

for k from 1 to *N* do

for i from 1 to *N* do

for j from 1 to *N* do

if edge[i][k] + edge[k][j] < edge[i][j] then

edge[i][j] 🡨 edge[i][k] + edge[k][j]

pr[i][j] 🡨pr[i][k]

end if

end for

end for

end for

return edge, pr

end algorithm

Key Operations for FloydWarshall: Comparisons and Addition

Complexity: T(n) = 2n3, n being the number of nodes represented in the matrix

The algorithm will iterate through exactly n3 times, performing one comparison and addition each time. This means best, worst, and average case is n3

**Algorithm** *getShortestPath* (int[][] pr, int i, int j):

*Input:* pr matrix as output by *FloydWarshall*, two integers representing start and end points

*Output:* List of integers representing the path from i to j

List<int> *path* 🡨 null

*path*.add(i)

while i != j do

i = pr[i][j]

*path*.add(i)

end while

return *path*

end algorithm

Key operations for getShortestPath: Append

Time Complexity: T(n) = n, where n is the number of nodes along the path from i to j

Best-case: Ω(n) = 1 : origin and destination are same

Worst-case O(n) = n : n is equal to the number of nodes in the graph, meaning

that the path went through every single node to get to the destination

**Algorithm** getAllPaths( int[][] pr ):

*Input:* pr matrix to pass to shortest path algorithm and control iterations

*Output:* 3-d integer matrix, shortestPaths

*length* 🡨 *pr*.length

*shortestPaths* 🡨 int[length][length][]

for i from 1 to *length* do

for j from 1 to *length* do

List<int> *path* 🡨 *getShortestPath*(i, j, pr)

If *path* != null then

*shortestPaths*[i][j] 🡨 *path*

end if

end for

end for

return *shortestPaths*

end algorithm

Key operations for getAllPaths: *getShortestPath* function call

Time Complexity: T(n) = n3, function iterates n2 times, calling a function of linear complexity

Best-Case: Ω(n) = n2 + n, where all paths are of length 1

Worst-Case: O(n) = n3, where all paths are of length n

Average-Case: Θ(n) = n3

**Algorithm** *populateTrafficMatrix* (int[][][] paths, int[][] flow) :

*Input:* 3-d matrix, paths, containing all shortest paths and 2-d matrix, flow, with all traffic flows

*Output:* 2-d matrix, trafficMatrix, containing the traffic load on all edges

*length* 🡨 *paths*.length

*a* 🡨null

*b* 🡨 null

*trafficMatrix* 🡨 int[*length][length]*

for i from 1 to *length* do

for j from 1 to *length* do

for k from 2 to *paths*[i][j].length do

*a* 🡨paths[i][j][k-1]

*b* 🡨 paths[i][j][k]

*trafficMatrix[a][b]* 🡨 *flow[i][j]*

end for

end for

end for

return *trafficMatrix*

end algorithm

Key operations for populateTrafficMatrix: Addition

Time complexity: T(n) = n3

Best case: Ω(n) = n2 + n, where all paths are of length 1

Worst case O(n) = n3, where all paths are of length n

Average case: Θ(n) = n3

**Program Sequence with Time Complexity:**

Given: int[][] edgeMatrix, int[][]flowMatrix, int[][] prMatrix, int start, int finish, int numberOfCities

FloydWarshall(edgeMatrix, prMatrix) T(n) = 2n3

getAllPaths(prMatrix) T(n) = n3

populateTrafficMatrix(shortestPaths, flowMatrix) T(n) = n3

FloydWarshall(trafficMatrix, prMatrix) T(n) = 2n3

getAllPaths(prMatrix) T(n) = n3

----------------------------------------------------------------------------------------------------

Total time complexity: T(n) = 7n3

**Supporting Data Structures:**

The only significant data structures that I used in this algorithm are arrays and ArrayLists. I only use ArrayLists in one portion of the algorithm, and for the rest of the algorithm I exclusively use arrays. Specifically, 2-d and 3-d integer arrays. This is done for a couple of reasons: speed and space.

Arrays are more efficient in access and memory than ArrayLists. This is because of their static nature. ArrayLists are dynamic in that they are able to resize when necessary, unlike normal arrays. This resize can be costly in performance, though. Seeing as the algorithm uses multiple n3 complexity functions, constantly resizing could potentially slow the program significantly. On the other hand, once declared, an array cannot change its size. This is the major tradeoff between the two structures. This also makes ArrayLists easier to program with, due to an increased functionality. There is only one time in the algorithm that the size of an array will be unknown (during the getShortestPath function call), and therefore that is the only time that I use anything other than arrays, despite the ease of using ArrayLists.

**Algorithm Modifications:**

I should discuss the addition to the Floyd-Warshall algorithm that allows for Path Reconstruction. It should be noted that this is not part of the default algorithm. Floyd-Warshall is strictly for traversing a graph and finding shortest distances for all pairs of nodes. The addition of the prMatrix allows us to recreate the path that Floyd-Warshall used to obtain the shortest distances. It is a small addition to the original algorithm, but it is absolutely crucial to solving the SneakyPath problem.

**Algorithm Implementation and Testing**

**Language Considerations:**

I chose to implement my algorithm using the Java programming language with the IntelliJ IDE. My first consideration was my own personal skillset. I will be honest and say that I did not even consider learning a new language to implement the program. This left me with the choices of Javascript, C++, C#, Java, or Python. Javascript seemed like a nightmare, and I haven’t used Python or C++ in quite some time, so for practical consideration, I was down to Java or C#.