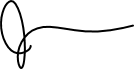
CS404, Fall Semester 2016: Find the Sneaky Path

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#.

Java and C# are very similar languages in many aspects. Java was built off of C++, and then C# followed Java. Both are designed for Object-Oriented Programming. Both enforce JIT (just in time) compiling, which can be a major performance booster when running large applications (i.e. looping the program thousands of times in order to get a time study). Speed and performance wise, both languages are fairly similar, neither having a dramatic edge. Both have extensive standard libraries and massive third party support as well.

C# has many advantages over Java, but none that factored into which language to use for this particular problem. C# has more extensive functionality, with features like Language Integrated Queries (LINQ) and Late-Bound (dynamic) Typing. If I were interacting with a database, or had to use queries at all, C# would have been my choice. When selecting which language to choose, it really came down to two factors: exception handling and familiarity. I have used Java more extensively, and am therefore more familiar with it. Java’s use of checked exceptions (and C#’s lack) was the other factor. Making exception handling more explicit was a great help to me, as it forces the programmer to know what type of exceptions will be possible at all points. Those two factors gave a slight edge to choosing Java for this problem.

**Testing and Validation:**

Validating the results of the finished program was one of the hardest parts of the project, especially since I did not produce the correct results right away. The first step after coding was inputting the original file given for the project and comparing my results with the results that were provided. My original code came up with similar results, but with a few discrepancies, namely that my Sneaky Path algorithm chose a slightly different path. This was alarming considering the nature of the project. After a lot of debugging, it turned out to be a problem related to Java being pass-by-reference. When I was making a copy of a particular matrix, I wasn’t copying the data, I was copying the reference, so the values contained weren’t what I expected them to be later in the program.

Once I had the same results as the given test results, I still needed more proof. I pursued four methods to ensure the program was operating correctly: performing the algorithm by hand, running the data through a third party calculation, running the data through third party calculation that uses a different algorithm, and comparing results with colleagues who used different algorithms.

Obviously completing the algorithm by hand was time consuming and still not trustworthy, but at the very least, when the results matched, it adds confidence that the program is running correctly.

For the third party, the website <https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html> (which is run by the same people from the University of Munich that I quoted in describing the Floyd-Warshall SPA) has a Floyd-Warshall simulator that runs through the algorithm step by step. Once I input all of the nodes and their weights from the Edge Matrix, I was able to compare the results. After a single pass, I input the values from the Traffic Matrix to ensure that the Sneaky Path was also correct.

The website <http://www.julianbrowne.com/article/viewer/shortest-path> provided a calculator to perform Dijkstra’s SPA, and recreated the path as well. This was particularly helpful, as well as confidence boosting when all results matched.

Lastly, I compared results with other students working on the project, but not using Floyd-Warshall as the underlying algorithm.

Once all of these preliminary comparisons were complete, I began to create test cases to run the algorithm against. Since a healthy amount of test data was provided for the project, I focused on a few edge cases. The first was simple. Input a graph such that all nodes in the graph lay along a single path, i.e. every node has a single edge to the next node until the end is reached. The results are mostly what they theoretically should have been (all paths being a subset of the only path through the graph), but there was a small predicament that gave me pause. There were paths going backwards (4 to 1, or 5 to 3) when the only paths I should have had were going forward. My input didn’t include any edges from a node to preceding nodes. The answer lied in the fact that my portrayal of INFINITY was simply a number large enough that the algorithm would never choose it. So all edges weren’t initialized to INFINITY, they were initialized as a very large, but real number. Given no other options, the algorithm chose the very large path. In retrospect, this is exactly the behavior I *should* have expected.

The next test case was using input where all weights on both the Edge and Flow matrices were of value 1, and every node had an edge to every other node. No real surprises here. Every path, both the shortest distance and the sneaky path, was a direct route from the origin to the destination. This is exactly how it should behave. The only interesting part, although not surprising, was after performing the time study on both cases, the single path test case was marginally faster than the test case where all paths were equal. This helps validate the best/worst case scenarios for time complexity that I analyzed previously.

**Insight from Implementation:**

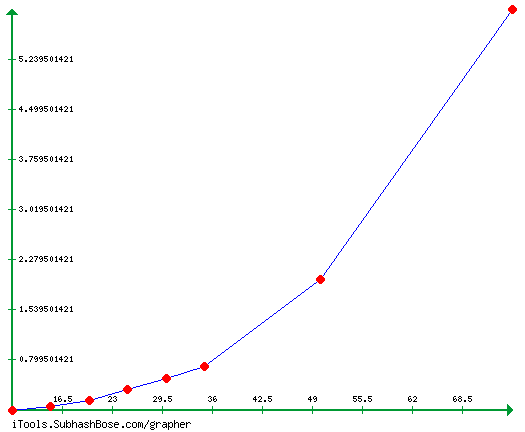
* I chose to use mainly arrays instead of ArrayLists or any other data structure do to their simplicity and advantages in speed and memory. When actually coding the algorithm, my getPath function, returns an ArrayList (the one part of the program that I deemed necessary to use that structure over and array), but ended up having to convert the List back into a simple array before I could add it to the ShortestPaths matrix. This nullified any real advantage I had hoped to gain in performance. It still cost less in memory, which is of significant importance, but I learned that it would have been easier to code, if I would have used ArrayLists the entire time.
* Off-by-one errors are always a problem. Instead of adjusting the values of the input matrices upon input and then change them back for output, I changed the values on my iteration variables. Originally, I saw the value of this being that I would never have to worry about the input and output not matching up, and when indexing into my arrays, it would be simpler. This resulted in numerous NullPointerExceptions being thrown during implementation, and a not insignificant amount of time being spent debugging them. “There are only two hard problems in Computer Science: cache invalidation, naming things, and off-by-one errors.”
* Always know the consequences of methods that you did not write yourself. As mentioned earlier, I ran into a problem with making a copy of a matrix. I thought that the copyOf() function performed a deep copy of the array, but instead it copied the reference. I ended up having to do a manual copy by iterating through the array and copying each value. Not a costly venture, but definitely a source of much frustration when I didn’t understand why my values were incorrect.

**Time Study and Analysis**

To compute the time it took for the program to complete the algorithm, I clocked the system time in nanoseconds before and after each key portion of the algorithm. In my output files, there is an elapsed time for each of the key functions, as well as a total running time at the end. These figures do not include the running time for operations that aren’t involved in the algorithm such as parsing file input or writing the results of the program. I did my best to only time the functions that were pertinent. That being said, there are a few points inside said functions that were not in the algorithm (such as matrix initialization or reinitialization), but were crucial to the programming aspect of the problem, and were included in the timing. They are all fairly small operations, and should not affect the timing much, but it seemed worth mentioning.

For the study, I ran each sample input 1000 times and averaged the times to try and get the most accurate times possible. The results are as follows:

|  |  |
| --- | --- |
| N | Time (in milliseconds) |
| 10 | .059501421 |
| 15 | .11736415 |
| 20 | .203139162 |
| 25 | .356811805 |
| 30 | .521376389 |
| 35 | .704272576 |
| 50 | 1.987994061 |
| 75 | 5.973645435 |



After plotting all of the data points and performing a cubic regression on the data, the resulting function came out to f(n) = 1.05n3 + 31.27n2 – 413.59n + 6995.7. This isn’t very helpful, but it does provide a little bit of insight. The graph obviously grows at an exponential rate, however it does not seem to grow at a rate of 7n3 as previously predicted. In fact, after examining the actual growth rate vs the predicted growth rate, it seems to be close to half what was expected.

Reasons for difference in expected vs actual results:

* The Java compiler is smart. Generally, the compiler optimizes the code as it compiles, getting rid of performance hits that would be caused by bad code. For example, if literally translated to machine code, doing a simple iteration such as i++ vs ++i makes a difference. The former is slower because it makes a copy of the variable and then adds one to it. The latter doesn’t do this. But after a modern compiler gets done with it, they both are the same. The point is that if I counted something as a key operation in my algorithm analysis, it could add an additional n3 to my calculations, when the compiler would have made it unnecessary.
* I overestimated my time complexities. By not tending towards worst-case scenarios for time complexity, I neglected the average-case. If the actual time complexity is something closer to 4n3 as the data suggests, this would account for a large deal of the discrepancy.
* My complexity for getting paths should have been a factor of n2 instead of n3. The function iterates through two for loops of length n. This was a mistake in my original analysis.

While the results are not exactly in line with my original analysis, I do find that they are within a reasonable degree of discrepancy. Running the tests and time study has reconfirmed that the Floyd-Warshall SPA has performances of O(n) = Θ(n) = Ω(n) = n3. Seeing as the Floy-Warshall portion of my own algorithm was the most expensive portion of the algorithm, it goes to prove that the SneakyPath Algorithm also has performances of O(n) = Θ(n) = Ω(n) = n3.

**Epilogue**

This project was both challenging and rewarding to complete. At many times I became immensely frustrated, but overcoming obstacles and achieving results was very satisfying. I learned many lessons, including:

* Don’t procrastinate! I know this is an obvious one, but it deserves saying.
* The importance of testing the algorithm before sitting down to code it. This is easier said than done, especially when you *think* you have the correct solution algorithmically and are eager to program it. Towards the middle of my programming, I ran into problems that backed me up numerous hours. I spent much time undoing the wrong that I had done, and all of it could have been avoided if I had spent more time going over the algorithm on paper before coding.
* Know the consequences of the code that you use. I mentioned this in the Implementation section, but it was an important lesson. Thinking the methods I was using did something other than what they actually did cost me even more time and effort.

If I were to do the project again:

* I would try writing it in another language to see the difference in both implementation and performance. Researching which language to use it one thing, but it is different to actually write the same program in multiple languages to see for yourself.
* I would run the program on different machines. I ran each input 1000 times to get proper results, but it did not occur to me at the time to test the program on different machines. It would be interesting to see if the results would have differed.
* I would use the same algorithm overall. I could rewrite it using a different underlying algorithm, but I was very happy with the way that my algorithm turned out. It was simple and the functions needed were fairly concise. Any problems with implementing the algorithm came from the actual coding, not the algorithm itself.

**Appendix A**

Program files:

* FloydWarshall.java (contains the implementation of the algorithm as well as the main program)
* MatrixCreator.java (contains the code for reading in the input files and initializing all the matrices needed to implement the algorithm)

Machine used: Acer Aspire R with Intel Core i7 processor

Language: Java

IDE: IntelliJ IDEA 2016.2.4