**CS404, Fall Semester 2016: Find the Sneaky Path**

I understand and have adhered to the rules regarding student conduct. In particular, any and 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.

Tu Vu 11/26/2016

**Introduction**

When looking for a direction, say from a to b, the majority of people would want to find the shortest route. There are several algorithms known that will help to find such path. However, this is not always the case. Since people tend to find the shortest route, it tends to be congested much easier. Some people may be interested in the route with the least traffic. They want to avoid other cars as much as possible. They do not want to inconvenience other people when towing a 12-foot wide boat, they do not want to see other cars, they want to find a peaceful route where they can drive without being stuck in traffic. This may be also of interest in the Computer Network: find the route with the lowest probability of dropped packets. This project is designed to find such path, and it will be called a *Sneaky Path*.

**Design and Analysis of the algorithm(s)**

The two main algorithms used in the program are Floyd-Warshall Shortest Path Algorithm (SPA) and Dijkstra’s SPA. The assumption of this project is that everyone will choose the shortest path to go from *a* to *b*. For example, say that there are F[21, 47] = 115 cars traveling on the path from node v21 to node v47, then there are 115 cars on each edge of that path. Suppose edge (v34, v38) is on the shortest path from node v21 to node v47, then there are also 115 cars/hour on the edge (v34, v38) due to the flow F[21, 47]. But is very well possible that the edge (v34, v38) is also on the shortest path from node v17 to node v29, and that F[17, 29] = 75. This means that there are at least 190 cars/hour on the edge (v34, v38).

The original ideas were to find all the path from the desired source and destination using modified Depth First Search (DFS) but that plan didn’t work out so well. Parts of the problem is that I don’t take in to account other people’s paths, which add traffic load on my path’s edges. Additionally, finding all the possible paths between two cities definitely does not perform well on a larger scale. I test it on the n = 75 cities, and it struggles to give me all the results in a reasonable time manner.

The second attempt was looking at Dijkstra’s SPA, but I need to find the actual carried traffic load from all source-destination pairs on each edge. The most suitable algorithm for this task is Floyd-Warshall SPA. Actually, Floyd-Warshall SPA is used to created the all-pairs-shortest-paths matrix. This combines with the Load Matrix (matrix L) to find the *Sneaky Path*. This will be explained as I go through the program in details below.

The program is split into 5 parts. I will walk through and explain each part of the program.

First, let me describe briefly the purpose and space complexity of each global variable used in the program:

* Size: this is the n number of cities or nodes in each case. This always has the space complexity of Θ(1).
* Source: this is the desired starting point of the *Sneaky Path.* This always has the space complexity of Θ(1).
* Destination: this is the desired destination of the *Sneaky Path*. This always has the space complexity of Θ(1)
* MyDist: the all-pairs-shortest-paths matrix. This always has the space complexity of Θ(n2).
* EdgeWeight: an adjacency matrix represents all the edges’ weight (matrix E). Each *(i, j)th*entry represents the traveling time on the (direct) edge between the two specific nodes, *i* and *j*. This always has the space complexity of Θ(n2)
* FlowMatrix: a flow matrix represents all the paths’ traffic (matrix F). Each *(i, j)th*entry represents the number of other cars that travel the entire path from node *i* to node *j* every hour. This always has the space complexity of Θ(n2)
* LoadMatrix: a load matrix represents all the edges’ load (matrix L). Each *(i, j)th*entry represents the total load from node *i* to node *j*. This always has the space complexity of Θ(n2)
* FirstStop: another matrix results from the all-pairs-shortest-paths Floyd-Warshall algorithm. This always has the space complexity of Θ(n2).
* SneakyPath: a collection which represents the discovered *Sneaky Path*. This has the worst space complexity of Θ(n) and the best space complexity of Θ(2). The worst space complexity happens when the *Sneaky Path* contains all the location. The best space complexity happens when the *Sneaky Path* only has 2 locations, the source and the destination.
* k: a multiple purpose counting variable. This always has the space complexity of Θ(1).

1. The first part’s main purpose is to look through the input file, and populate all the necessary global variables for this program. There is a *foreach* loop which will loop through each line of the input file, identify which matrix the data belongs to, and populate the following: *MyDist, EdgeWeight and FlowMatrix*. The best and worst time complexity of the for loop is Θ(t), where t is the number of lines in the input file. The space complexity of this part was accounted in the variable section above.
2. The second part is where I am looking for the all-pairs-shortest-paths matrix. Obviously, the suitable algorithm for this part is the Floyd-Warshall SPA. The heart of the algorithm is the three *for* loop. The algorithm computes the shortest path for all *(i, j)* pairs for k from 1 to k (the number of cities). In each iteration, the algorithm compare the current distance of *(i, j)* with the distance through an alternative node *k (i, k, j)* in *MyDist* matrix. Additionally, it will update the destination node that each shortest path needs to go through in the matrix, *FirstStop*. *FirstStop* matrix is the guiding matrix for each *(i, j)* pair shortest path. With this matrix, we are ready to go to the next part. The time complexity of this algorithm is Θ(n3) in both best and worst case. The algorithm has 3 *for* loop that loops the same number of times as the number of cities. It cannot avoid doing stuff in any iteration, so it has the same best and worst time complexity. There is no significant space complexity besides temporary variable which is Θ(1) each.
3. This is where I calculate the load on each edge of the graph (matrix L). I do this by looping through the Flow matrix (matrix F). On each *(i, j)th* entry of the F matrix, I see if there is a stop on the shortest path or not by using the *FirstStop* matrix. If there is a location that leads to somewhere, this means I’m not done with the current shortest path. I need to add the load of the shortest path (from matrix F) on *(i, j)* to each of its edge. In the end, I have a matrix that has the load of all the edges in the graph. It is ready to be used in the next part. The best and worst time complexity of the algorithm used in this part is Θ(n2). The algorithm has to loop through the F matrix with size n\*n. I cannot imagine it being cheaper or more expensive. There are temporary variables being used, but each of them is Θ(1) in space complexity. It is quite insignificant. Besides that, the rest of the work is value reassignment. There is no additional space complexity.
4. This part is where I calculate the *Sneaky Path* using Dijkstra’s DPA. Since I have already known the load on each edge in the L matrix, the rest of the work is just finding the path with the least load. This is what I use Dijkstra’s for. Dijkstra’s DPA is designed to calculate a shortest path from one node to the rest of nodes in the graph. One other possibility is using Breadth First Search algorithm (BFS). However, BFS is not designed to work well with a weighted graph. One can use a modified BFS, but I will just use what’s available to me, Dijkstra’s DPA. In my implementation, the priority queue I used to store the discovered nodes is a binary min heap. The heap is not my design. I use a third-party, open source collection library for C#/.NET called C5. I decide to go with a third-party implementation to reduce the complexity of the program and save that implementation time for other purpose of the project. I personally don’t think it is worth the time to re-implement and test a heap implementation given the time constrain of the project. My implementation is similar to the implementation on the Graph handout in this class. One minor modification is using boolean value instead of color to mark a visited node. With the min-heap as the priority queue, the time complexity of the algorithm is Θ(2(n + m)\*lg(n)). With this heap has multiple copies of a node, the heap space complexity is n2 elements, and 2\*lg(n). Additionally, there are 3 other array with size n to keep track of visited nodes, distance from the source of each node, and the parent of each node in the path.