There were three main options to take into consideration when approaching this problem: Dijkstra’s, Floyd-Warshall, and taking the power of each adjacency matrix. Taking the power approach is simply multiplying the adjacency matrices. From there any non zero value indicates a path of hop size = the power of the matrix. For example say we have matrix E that gives the travel time from any node in the system to another E x E or E2 would give all the paths of hop size 2 from any node to another. It is important to note that this approach only really yields paths that exist and does little to consider finding the most efficient route from one node to another. Infact, doing so doesn’t even yield the correct “travel time” between the two nodes, it only tells us that there exists a path of length 2 from one node to another. This would eliminate the option of even taking the minimum of the resulting values of each matrix multiplication to find the shortest of the options yielded. The only information we would have is that we know there exists paths of varying lengths from one node to another. We could just choose the “shortest” path in terms of the smallest hop count but this isn’t likely to yield great results.

This leaves us with the two other options of Dijkstra’s and Floyd-Warshall. From a general perspective this is primarily choice between dynamic programming and greedy approaches. It is important to note that both of these approaches require the assumption of the optimal substructure property. This means that the problem can be broken into smaller subproblems with similar structures in which if we can optimally solve them will eventually yield an optimal global solution.

However, one potential criteria we could consider in evaluating the choices is difficulty in implementation. Dijkstra’s presents a clearly easier implementation as we only need to survey the surrounding nodes, choose the node with the smallest weight of nodes we have visited and repeat until we reach our destination node. For Floyd-Warshall we calculate the shortest path by considering every other option to that node and compare it to the currently known shortest option. If we discover traveling through another node first and then going to our destination is “faster” then we will set the new travel time as the shortest known path. While the implementation may not be substantially more difficult it does require more options to consider and things to keep in mind. In this regard we will assume Floyd-Warshall to be harder to implement. But as stated before we will not factor in the difficulty of implementing because we are looking for the best solution with best meaning in terms of results and efficiency.

In regards to efficiency of computation Dijkstra’s stands out as being more efficient. For one, once we reach our desired destination point we stop computation. This is a clear computational advantage on Floyd-Warshall which computes all options between each node and then determines the shortest path to take. Additionally, Dijkstra, for any two nodes, will not consider a shorter path through every other node and take the minimum from there as FW does, instead it chooses the local minimum. This obviously is less computationally expensive but has its trade off in final results. When it comes to space Dijkstra also has an edge as FW needs to store all of the past computations that is used for later comparisons whereas Dijkstra only needs to store the nodes it has visited and needs to visit. In comparison along use of resources Dijkstra clearly has the edge when it comes to efficient use of resources. However, our final and most important consideration for this project is the final result.

As alluded to earlier, Floyd-Warshall is more likely to lead to a better result in terms of the solution. This is because FW uses more computational resources to evaluate all of the options between two points. By solving this for every node we are more likely to get the best answer possible. The problem with Dijkstra, in these terms, is that Dijkstra is always choosing the locally optimal solution. The problem arises when there exists an alternative route that may initially be less efficient but when we look at the global result it ends up being more efficient. Although Dijkstra is easier to implement and less computationally expensive, we chose to implement Floyd-Warshall in order to obtain a more complete solution for both parts of the problem. The other advantage that Floyd-Warshall approach has is it solves the problem for any given two points. Since it has to calculate all of the dependant subproblems when we reach an answer we also have the solution for any other 2 points in the system. It is important to note that Dijkstra does not allow for this and if the two points were to change we would likely have to recalculate.

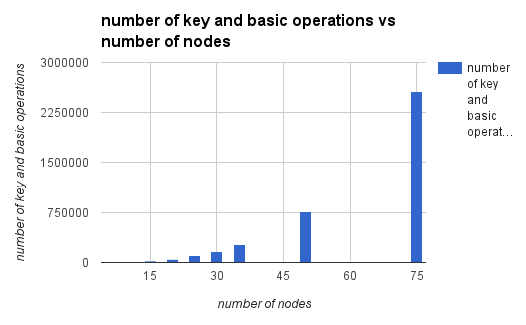
The problem statement is as follows: given a network of n connected nodes and two n x n matrices where one gives the “travel cost” between any two nodes in the system and the other gives the “flow” or number of cars traveling in the direction from one node to another, find the path between a given start node and a given end node such that the number of cars you encounter traveling in the same direction is minimized. If we were given a matrix with the number of cars traveling on each “road” or edge we simply would just have to apply one of these shortest path algorithm with the number of cars on each edge being the weight. However, because we are not given that matrix we must use the matrices we are given in addition to a simplifying assumptions in order to arrive at our answer. Because we know how long it takes to travel from one node to another on a given “road” and because we know how many people are traveling from point a to point b we make the assumption that each person will calculate the shortest travel time between to points and take that path. It is important to note that in the “real world” this assumption is approximate but does not always hold. For instance there may be other people that also want to take the “sneaky” path and not find the shortest travel time between two points. Or people who are really cheap and would rather save $10 in gas taking the shortest distance path even if it is at the cost of more of their travel time. Another assumption we make is that the travel time is fixed regardless of how many people are traveling on this road. This is another assumption that doesn’t hold, for evidence please see highway I-35 at literally any point in time in the week because why not have bumper to bumper traffic at 10pm on a weekend.

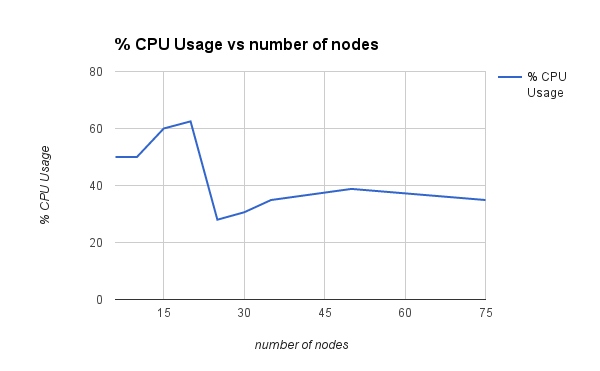
After we have calculated the shortest travel time between any two points (as stated before we use Floyd-Warshall for this) we use this information to construct the matrix we originally needed for the sneaky path calculation (i.e. the matrix that gives us the number of cars on each edge traveling in a specific direction). We achieve this by stepping through the shortest travel time path between every node and adding the number of cars traveling on that path to the edge traffic matrix for each edge it transverses on its path. Once all of the paths have been considered we will have a complete matrix with all of the edge traffics. We will once again find the “shortest” path using these edge weights (and again the Floyd-Warshall algorithm). However since these weights represent the number of people traveling on them the resulting matrix is the “sneaky” path. For this solution we chose to use essentially a 2d array structure. This was for simplifying the implementation and because we assumed the given network of points to be sufficiently densely connected meaning there are many paths one could take between any two points. If this is not true on the given network of points and roads one would be better off using a 2d linked list. The advantage of this approach is you are not storing data to indicate a road doesn’t exist between two points. This saves time and space as you aren’t storing the data and you aren’t performing any operations on it only to yield non-meaningful results (saving computational time). This is an obvious area for improving the algorithm if desired.

This algorithm, even with our simplifying assumptions, does have room to result in differing sneaky paths. Namely the case when we have two paths that result in the same edge weight. For instance say (2,3) = 5, (3,5) = 4, (2,5) = 9. Then our algorithm has to choose between taking the path 2 -> 3 -> 5 = 9 or 2 -> 5 = 9. Both give the same resulting travel time but will lead to a different path. If you are only applying the algorithm once this may not be of much consequence. However, in our situation the path that the algorithm takes will change the results for the edge traffic for (2,3), (3,5), and (2,5) depending on which path is taken. Since we use these values to calculate the “sneakiest” path then the resulting final sneaky path that is taken will also be affected by this decision. Since the second time we apply Floyd-Warshall it uses those values as the assumed amount of traffic it will still give the “sneakiest” path based on the information it was given. However, this is a demonstration of even by using the same algorithm and assumptions we can arrive at varying results depending on the specific implementation details.

As far as the performance goes, the space usage is going to be O(n3). This is primarily from all the the matrices that are stored. There is the 2 initial matrices that are given along with n copies for each iteration where a new node path is tried, 2 matrices to keep track of the next node to visit for the optimal path and their n copies for each iteration 2(n+1). Then two matrices to store the path for each node to another and an edge traffic matrix to store the intermediate result between applying the two algorithms (discussed above). This comes to 4n3+7n2 or O(n3). As for the time complexity for the best case it would be O(n3). This is because each algorithm would run around (3n)3 and the path construction matrices would run around (4n)2  and (8n)2 respectively. For worst case on the other hand, the algorithms would run at the same time complexity and not be affected at all. However the path construction would move to neighborhood O(n4) assuming every path has to go through every node. This may not be a likely scenario since the goal is to find the sneak path but for the purposes of algorithmic analysis we will allow for this case. These complexities are under the assumption of key and basic operations being assigning values to an array and additions but not counting temporary variables used for convenience/clarity of implementing the algorithm. Both of the time complexities did not take into consideration the time to construct the arrays for storing the data.

Below are experimental data showing the performance of the system as a function as the number of nodes (n). As you can see roughly speaking the number of key and basic operations are exponential in nature. The values ended up lying between n3 and n4 so the analysis seems to be accurate. We also see some unexpected behavior with the CPU. My guess was the percentage of usage would be non-decreasing. However the results show a dip once we cross 25 nodes. This is likely explained by the the work being distributed across more cores as the workload increases.





|  |  |  |
| --- | --- | --- |
| number of nodes | number of key and basic operations | % CPU Usage |
| 6 | 1492 | 50 |
| 10 | 6715 | 50 |
| 15 | 21531 | 60 |
| 20 | 50617 | 62.5 |
| 25 | 98108 | 28 |
| 30 | 166792 | 30.6 |
| 35 | 265000 | 34.9 |
| 50 | 763038 | 38.8 |
| 75 | 2560026 | 34.9 |

So as far as design and data structures goes I think I am pretty happy with my decisions. It could have been more efficient if I had used 2d linked lists instead of arrays but I don’t think the increase in efficiency would have been worth my time. I am happy I used the Floyd-Warshall algorithm as it will yield better results and I was able to get a better understanding of dynamic programming as a result of this exercise plus after covering Dijkstra for the second time I don’t think it would have been as interesting to implement. As far as unforeseen situations I wouldn’t say that there were many that came to mind. If anything it was more of clarifying what exactly we are being asked to do. When it came to that just asking questions in class and sending a couple of emails clarified any uncertainties. Having the example to work off of was also a HUGE help. If I didn’t have the example I’m not sure if I would have been able to complete the implementation and may have just went the easier route and done Dijkstra. It was helpful to be able and clarify what we were being asked to calculate. As far as what I have learned I think it has been a couple of things. First I think I do have a better grasp of taking a problem and assessing the best algorithm to solve the problem. I think I also deepened my understanding of dynamic programming and how it differs from greedy approaches. If I had to do this project over I’m not sure if I would change anything. If there were additionally requirements I may not have done the more challenging implementation depending on how much harder the constraints would make it. I don’t know if I’d say I had deep longer lasting take aways where I found myself (or Jesus) or have anything to pass on to future generations. I would just say it's just something you need sit down and do. I consider this achievement another small but nonetheless important victory in the endless fight against communism.