Tanner Siebe

CSCI 406

1/30/2017

Traveling Sales Person (TSP) Project

1. This project required two implementations to complete. But to get started I first started building around my input methods, there were two methods used one was reading in from a file input while the other was using a random number generator and placing them inside of a vector. The random number generator allowed for the number of points to varied and adjusted quickly to test out the algorithm that had been developed. Now as one of my methods, the nearest neighbor method changed the data to work through it I had to create various copies of the vector. Now onto the actual methods used.

The first method was the nearest neighbor method, this method works by starting at a point then finding the closest neighbor to it and moving to that point. This process is repeated until all points have been traveled to and then returns to the start point. To complete this method, I passed in a vector into the function, then initializes a variable called length to zero to keep track of the distance, as well as starting an output stream to a text file. After this a new vector was created to store the tour path using this method. The start point is put at the start of the tour vector and is removed from the vector containing all the points that need to be visited during the course of the travel. Once this has initially happened a while loop is utilized while there are still unvisited points. Then inside of the while loop a for loop is called to go through all remaining points. Each time through the for loop the distance between our current point as well as the point in question in the unvisited points vector is calculated. Then this value is compared to our minimum distance value. If the distance is less, then the index is saved as well as the minimum distance is set to that distance. Once it has gone through all the points that have not been visited the minimum distance will be holding the value of the minimum distance, this is added to length, the point itself is then removed from our points vector and is added to our tour vector. This is repeated until there are no unvisited points left due to the while loop. After the while loop is completed we then add our start point onto the very end as we have to return to it. Then we total our distance going through the tour vector from point to point. Following this we then output the length to our output file, as well as the points in the order that they are visited. This method is very quick but is not always the shortest path.

Now for the exhaustive search. This method works by setting one point as a start point and generating every possible path through every point then comparing these lengths to one another and finding the shortest of them. The function calls for a vector of our points to be passed into it, this vector will either have been created artificially using random numbers or read in from a file. This is accomplished in an implementation by creating a vector that will hold our optimized path as well as an output file that the values will be sent out to following completion. After this a vector of vectors is created to hold all the different permutations of our original vector. Then the optimum vector that will ultimately hold our shortest path is set to the vector that is passed into the function. Then utilizing a do while loop and a built in function of C++ ever permutation of our vector, aside from the first element that is our starting point, is created and added to this vector. After this a for loop is used to go through every vector in the vector that holds every path. This gives access to each path. Another for loop is used then to work through each vectors points. For each vector the distance from each point to the subsequent point is calculated and added to a length placeholder length. This length is then compared to the optimum paths length, if the length is less than the optimized path’s length then the vector that is shorter takes its place and the shortest length variable is updated. This process is repeated until every permutation in the vector of vectors is worked through and tested.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number Of Points | Time | Number Of Points | Time | Number Of Points | Time | Number Of Points | Time | Number of Points | Time |
| 5 | .004 | 50 | .00667 | 150 | .0133 | 600 | .10833 | 3000 | 2.391 |
| 10 | .005 | 60 | .007 | 200 | .01833 | 700 | .145333 | 4000 | 4.300 |
| 15 | .00533 | 70 | .00733 | 250 | .026 | 800 | .18633 | 5000 | 6.487 |
| 20 | .005667 | 80 | .008 | 300 | .034667 | 900 | .230667 | 10000 | 25.51 |
| 30 | .006 | 90 | .008667 | 400 | .054 | 1000 | .283667 | 15000 | 57.39 |
| 40 | .00633 | 100 | .009 | 500 | .07933 | 2000 | 1.084 | 20000 | 101.1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number Of Points | Time | Number Of Points | Time | Number Of Points | Time |
| 1 | .005 | 5 | .011667 | 9 | 1.562 |
| 2 | .005667 | 6 | .01367 | 10 | 15.381 |
| 3 | .00633 | 7 | .0413 | 11 | 165.732 |
| 4 | .01033 | 8 | .204 |  |  |

\*Each time listed is an average of 3 runs.

1. For the nearest neighbor approach, I found the worst-case time complexity came out to be O(n2). I decided to test this premise by timing 3 runs of the implementation then averaging the times. Then I increased and scaled up the number of points to see how the larger input into the function impacted the runtime. Overall, we see the trend follow a n2 fit, there was an attempt to model a line on the graph itself but when adding the more extreme cases this lead to some issues graphing the trendline. When this trendline was added the constant on the x2 term was nearly zero, as this is the case we could easily add a constant coefficient to scale the data that would hold it as an upper bound for all values observed. But looking at the data and comparing overall data points as if they were scales of each other we can see how this relationship plays out. This can be seen when comparing runtimes that use a number of inputs that are multiples of one another or are scaled up, we will look at multiple examples of this. The first example we will look at is the 500-point example, which we will show as O(m2), with m representing the size, when we compare this to 1000-point we would have O((2m)2), this leads to 4O(m) when some algebra is utilized. As we can see this follows the theoretical calculation based on the times observed. This trend can also be seen when looking at increasing from 10000 points to 20000, it follows nearly exactly the relationship we would expect for an n2as the 20000 is nearly four times that of the 10000, which if we compare them on the same scale in terms of a variable say m we would see that the 20000 would be roughly 4m when put into our O(n2) as compared to our runtime for 10000. This further supports our theory that we could have an upper bound if we added a constant coefficient, in theory anything over one would work, but for the sake of being a true upper bound we could go higher and use a constant such as five. With this constant it would always serve as an upper bound.

Now for the exhaustive search method I proposed the worst-case complexity was O(n!). This proposal was tested using the same averaging system as before and supported it with showing the times in relation to how many points were being used in each run. With this method, we can see just how quickly this method blows up in relative time to complete its function. After graphing the data collected it became a challenge of finding a best fit for the data, the highest degree that was able to be modeled was x12 and even then, it failed to grow rapidly enough. With that being said an exponential fit was able to model somewhat closely in the lower end of the data field where the amount of points was less than seven but after that the function could no longer accurately represent the time it would take for each run. So moving forward with this knowledge I moved to X!, but this is not an easily graph function when it comes to scaling to this so I used a scaling method to compare the points to one another and found that when scaling so there was no distortion of data that the X! fit was always safely above the true data. This implies that is a worthy assumption for the worst-case complexity. We can see this assumption showcased with the data obtained, let us look at the data for both the 10 and 11 point examples. With a O(n!) we would theoretically expect that the 11-point example would be 11 times slower than the 10-point example based on the worst-case scenario. When looking at the actual obtained value we get nearly exactly this as the runtime for 10-points was approximately 15 seconds while the runtime for 11-points was 165 seconds. This supports our theory that it is O(n!). For this small set of data points when a trendline was fit the coefficient was nearly one but to serve as a better upper bond we could add a constant coefficient of three that would hold and when combined with our trendline of n! would meet the requirements for the upper bound for all data points observed.

1. Now for some of the logic behind the methods used in this project. As for the nearest neighbor the amount of points used was a simple choice as it allowed to see how the function worked at a small scale and we see relatively little impact on the overall runtime. But it was still a good idea to know how it worked for the lower amounts, then as the experiment was scaled up values were chosen so that they were scalar multiples of one another and had that key relation so it could be seen how the program behaved. The nearest neighbor by comparison to the exhaustive search was a very quick running program so this allowed for more values of points to be tested to paint a clearer picture on what was happening. With these data points of various sizes, we can see the relationship that we would theoretically expect. We see that when we scale one data point in terms of another, scalar multiple, that it does impact the runtime as would be expected, in that the runtime of the larger input is roughly the scalar multiple squared the time of the original runtime. That is the behavior we would expect with this complexity.

As for the exhaustive search the values chosen for the amount of points was simply out

of necessity as it was not feasible to test with anything over 12 to 15 points depending upon how powerful of a machine was being used. So, it was decided to test up till where the time it took to complete was still manageable as each would be run multiple times for the averages. This time could have been reduced through further optimization, but given the constraints of the machinery it would have resulted in negligible gains. As for the behavior witnessed versus the expected behavior when we scale the inputs in terms of each other we get the exact response we would expect. This is again repeating the same behavior and method as before as modeling an input in terms of another as a scalar multiple and comparing it in our O(n!) notation. We can see the trend in our data as well as graphically that it follows the behavior as expected. It would be very interesting to see how this behavior continues as more and more points are added as well as program optimization. But in the scope of this class and project the method performs and follows the expectations we have for something of this nature.