Christopher Bonin
CSCI 406

Traveling Sales Man Project

Explain the details of your two implementations. Specifically, in both cases, discuss how you efficiently implemented high-level "english" statements provided in the pseudo-code.

With the **Nearest Neighbor implementation**, I start with picking the first point that is in the array. From there I do a while loop until the array has been emptied. At the beginning of the while-loop, I calculate the distance of the next point and the current point, then it goes to a another for loop with the array. It compares all the other points distances to the current distance to the next point. If It finds a point that it's distance is smaller, then I move to that point. I do this until all points have been removed from the array.

With the **Brute Force Exhaustive Algorithm**, I start with the permutation of ALL possible combinations of permutation for the route to take. I then go into a for loop of ALL permutations. I then go into another for loop that calculates the total distance. I assign that to a dictionary with a key value pair. Once the loops are done with all the distances of the best paths, I then go into another for loop that compares all the distances, and I choose the one that is best.

Determine the worst-case time complexity of your algorithms in terms of n. (This will depend on your implementation.)

The Nearest Neighbor: because of my usage of the while loop and the double for loop inside the while loop, It causes a bit of nasty worst-case time complexity. The while loop will execute n!. The inner for loop will run n times. The worst case complexity will be **O(nn!)**

The follows the same strategy with the **Brute Force Algorithm.** Because of the double for loop inside each other, and with the permutations iteration, we can see a worst case complexity of **O(n*n!)**

Use a random number generator to devise inputs for your algorithms for at least four different values of n.

NearestNeighbor:

n	Execute 1	Execute 2	Execute 3	Average
5	6.890296936035156e-05	6.890296936035156e-05	8.487701416015625e-05	0.00007422765096
10	0.0001380443572998047	0.0001380443572998047	0.00013709068298339844	0.0001377264659
50	0.0012822151184082031	0.0010929107666015625	0.0014228820800781255	0.001266002655
1000	0.32436490058898926	0.3189070224761963	0.3332788944244385	0.31833600997924805

Brute Force:

n	Execute 1	Execute 2	Execute 3	Average
5	0.00084710121154785 16	0.00075817108154296 88	0.00080084800720214 84	0.0008020401001
7	0.02672290802001953	0.02563905715942382 8	0.02713608741760254	0.02649935087
9	2.4816019535064697	2.5519449710845947	2.497218132019043	2.0891481374
10	29.63512682914734	27.825818061828613	28.308568000793457	28.5898376306

Nearest Neighbor n - values:

With the values of n, I wanted to try different complexities. Due to the nature of my computer, I wasn't quite sure if it would survive. So I played it safe first with values of 5 and 10. Just doubling the result. Then I jumped it up to 50 with the mindset of it being 5 times "slower" than 10. I then multiple 50 by 20 and decided to use 1000 to see it's full potential with this Nearest Neighbor Algorithm. With comparing the averages, you can see the exponential jump in time every time we move n's.

Brute Force n - values:

The complexity of Brute Force: I decided to use 5, 7, 9 and 10. I do not think that these were the optimal n's to choose but they were the best without destroying my computer.

Match theory and practice: Argue/demonstrate that your experimental runtimes are consistent with the theoretical complexities you derived.

As we can see with the results of the **Nearest Neighbor**, it is a lot faster than what I proposed the worst time complexity. From 5 to 10 times, we can see a 6 times difference in slower speed. In 10 to 50, which I'll remind you was an increase of about 5 times instead of 2 like the last one, we get a decrease of about 100 times (i.e: .00013 to .0013s). We then see the final result from 50 to 1000 which has an 60 times increase. So interestingly enough, we can see that it has a complexity of $\mathbf{O(n^2)}$.

With the **Brute Force,** we can see the complexity is spot on for **O(n*n!).** We get about a 60 times slow down in time due to amount of n. From 7 to 9, we can see 100 time increase (0.026s to 2.48s) for the result. We then can see another 100 times increase of time (2.48s to 29.65s). With this being said, you can see that it is exponentially growing each time with an increase of n.