Project 1: TSP-Timing Report

Phi Stanton

CSCI 406: Algorithms

Project Description:

This project had us implement two different approaches to solving the Traveling Sales-Person problem. The first one to implement was the "Nearest Neighbor" approach, which in brief would create a tour by selecting the nearest unvisited point until all points were visited. The second approach implemented was the "Exhaustive Search," which takes all possible tours and selects the shortest distance from it.

Nearest Neighbor Implementation:

For the nearest neighbor implementation, my code ran through a recursive function, that was handed a 2d-List of all unvisited points, the current point, and the starting point. As it progressed through the list, it would remove the current point from the list of unvisited points, and choose the next point by whichever was closest. Once the only point left in the list is the current point, the function calculates the distance between that point and the starting point, and passes that value upwards to add to the total distance of the tour. In the case of ties, the point that is closest in the list is the point that is selected.

This implementation is efficient, as by removing the points from the list, distance comparisons are not repeated, with the exception of the final point needing to compare the distance with the starting point again. By using a 2d-List, the individual points can be stored as integers instead of as the strings they came in as, which allows for much easier calculation of distance. (See Appendix A. Fig-3)

Worst-Case Time Complexity:

nearest_neighbor(): O(n)

("i" = number of times the recursive function was called)

list.remove(x): Called *n-i* times = O(n)

For Loop: Called *n-i* times = O(n)

calc_distance(): O(1)

Results in: O(n*2n) -> O(n^2)

Runtime on "n" inputs:

The range of n was chosen, starting at n=900. This is because the recursion depth in python is too great at n=1000, so I rounded down to the nearest 100 in order to get the maximum value of n to use. From there, the remaining n's were chosen by halving the previous value, down to n=56, which is well above the resolution of the clock (316ns).

By spreading out n over a large range, it makes it much easier to verify the theoretical runtime, as the large range of numbers will make the graph better to read.

N (Resolution of 316ns)	**Trial 1 (Seconds)**	**Trial 2 (Seconds)**	**Trial 3 (Seconds)**	**Average**
900	0.408799	0.459057	0.489551	0.452469s
450	0.132044	0.119680	0.110711	0.120811s
225	0.034690	0.041687	0.034905	0.037094s
112	0.013534	0.014870	0.013109	0.013837s
56	0.007828	0.006974	0.008850	0.007884s

Verifying the Theoretical Runtime:

The graph below is a representation of how different values of n affect the runtime. By looking at the shape of the graph, it follows the trend of an O(n^2) implementation, verifying the previous analysis of the algorithm.

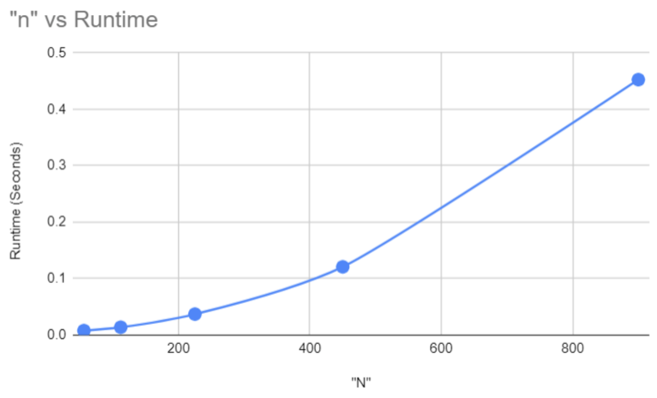

Figure 1: N vs Runtime of Nearest-Neighbors Approach

Exhaustive Search Implementation:

For the Exhaustive Search implementation, my code took a single starting point, and calculated the tour length for each permutation of that point. All of the points aside from the starting point are stored within a 2d-List, and by using the permutations() function, all permutation of the remaining points are generated. Looping through these permutations, the total distance between the points is compared to the previous minimum distance, replacing it if it is less than. Once all the permutations have been walked through, the minimum distance at the end is returned.

This implementation is much less efficient than the heuristic, but it is guaranteed to return the shortest tour possible. A few steps were made to reduce the time constraints on this function. Because the tour is cyclical, we only need to calculate the tours from a single point rather than starting from every point within the tour, drastically reducing the time at higher values of n. (See Appendix A. Fig-4)

Worst-Case Time Complexity:

```
exhaustive_tour(): O(1)
    list.remove(x): O(n)
    permutations(x): O(n!)
    for permutation in permutations:  O(n!)
        for i in range(len(permutation): O(n)
            calc_distance(x): O(1)
```

Results in: O(n + n! + n!*n) -> O(n!)

Runtime on "n" inputs:

The range of n was chosen starting at n=10, as I first had to verify that the runtime of n=10 was less than 10 seconds. Once that was verified, values around 10 were selected, but not far from each other as the predicted runtime was O(n!). The minimum value of n gave a runtime much greater than the clock's resolution (316ns), making sure that the runtimes calculated were accurate.

Because the time complexity of this algorithm is O(n!), subsequent values of n were chosen in order to be able to verify the complexity more easily.

N (Resolution of 316ns)	**Trial 1 (Seconds)**	**Trial 2 (Seconds)**	**Trial 3 (Seconds)**	**Average**
11	49.113564	51.517779	48.043544	49.55860
10	4.683638	4.617638	4.604584	4.635286
9	0.501588	0.508950	0.448236	0.486258
8	0.059269	0.060837	0.056326	0.058811
7	0.011984	0.011969	0.013963	0.012638

Verifying the Theoretical Runtime:

The graph below is a representation of how different values of n affected the runtime of the program. The value n=11 was removed in order to get a better view of the earlier values within the graph. This graph follows a complexity of O(n!) fairly closely. The difference between individual values seems to be a scalar of each other. For example, the proportional runtimes of n=10 and n=11 is 4.64:49.55 or 1:10.67, which is close to the 1:11 proportion expected of an algorithm with O(n!) complexity.

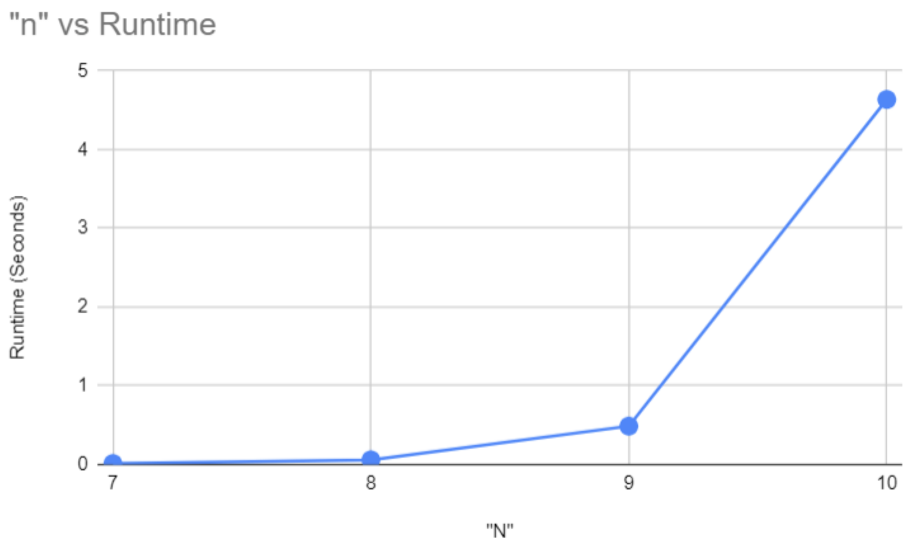

Figure 2: N vs Runtime of Exhaustive Search Approach

Appendix A:

Figure 3: Python code for Nearest Neighbor Approach

```python
import math
import sys
import time

#Read file, give me n and also 2xn array for all the points
def read_input(fname):
    f_input = open(fname, "r").read().split('\n')
    n = int(f_input[0])
    input_t = f_input[1:]
    input_a = []
    #split input into 2d array
    for i in range(n):
        input_a.append(input_t[i].split())
        #change from string to int
        for j in range(2):
            input_a[i][j] = int(input_a[i][j])
    return input_a

#distance formula
def calc_distance(point_1, point_2):
    p1 = point_1
    p2 = point_2
    d = math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
    return d

#input_t is the input, i is which point its visiting
def nearest_neighbor(not_visited_list, p, initial_point):
    #distance back to original
    if len(not_visited_list) == 1:
        return calc_distance(initial_point, p)
    else:
        not_visited_list.remove(p)
        min = calc_distance(p, not_visited_list[0])
        closest_point = not_visited_list[0]
        for point in not_visited_list:
            check_dist = calc_distance(p, point)
            #using < to stay with the first input in case of tie
            if check_dist < min:
                min = check_dist
                closest_point = point
        return nearest_neighbor(not_visited_list, closest_point, initial_point) + min
```

```python
def main():
    start_time = time.time_ns()
    if len(sys.argv) != 1:
        input_t = read_input(str(sys.argv[1]))
    else:
        input_t = read_input("input-900.txt")
    not_visited_list = input_t
    #start at the first point given
    nn_tour = nearest_neighbor(not_visited_list, not_visited_list[0], not_visited_list[0])
    #rounding
    nn_tour = round(nn_tour, 3)
    print("%.3f" % nn_tour)
    runtime = round((time.time_ns() - start_time)*(10**-9), 6)
    print("Time in seconds: " + str(runtime))

    return 0

if __name__ == "__main__":
    main()
```

Figure 4: Python Code for Exhaustive Search Approach

```python
import math
import sys
import time
from itertools import permutations
#Read file, give me n and also 2xn array for all the points
def read_input(fname):
    f_input = open(fname, "r").read().split('\n')
    n = int(f_input[0])
    input_t = f_input[1:]
    input_a = []
    #split input into 2d array
    for i in range(n):
        input_a.append(input_t[i].split())
        #change from string to int
        for j in range(2):
            input_a[i][j] = int(input_a[i][j])
    return input_a

#distance formula
def calc_distance(point_1, point_2):
    p1 = point_1
    p2 = point_2
    d = math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
    return d

#temp_list is copy of original list, start_index is the index its started at
def exhaustive_tour(temp_list, start_index):
    start_point = temp_list[start_index]
    points = temp_list.copy()
    points.remove(start_point)
    #get all permutations of the points
    min = sys.maxsize
    for permutation in permutations(points):
        current_distance = 0
        for i in range(len(permutation)-1):
            #get the first side
            if i == 0:
                current_distance += calc_distance(temp_list[start_index],
permutation[i])
            current_distance += calc_distance(permutation[i], permutation[i+1])
        #get the last side
        current_distance += calc_distance(temp_list[start_index],
permutation[i+1])
        #compare distance to the minimum distance
        if current_distance < min:
            min = current_distance
    return min


def main():
    start_time = time.time_ns()
    if len(sys.argv) != 1:
        input_t = read_input(str(sys.argv[1]))
    else:
        input_t = read_input("input-11.txt")
    ex_tour = exhaustive_tour(input_t.copy(), 0)
    #round
    ex_tour = round(ex_tour, 3)
    print("%.3f" % ex_tour)
    runtime = round((time.time_ns() - start_time)*(10**-9), 6)
    print("Time in seconds: " + str(runtime))

if __name__ == "__main__":
    main()
```