Maria Deslis

CSCI406-SPRING2013

TSP PROJECT

Professor Mehta

**1. [15 pts] Explain the details of your two implementations. Specially, in both cases, discuss**

**how you efficiently implemented high-level \english" statements provided in the pseudo-code.**

1. **Nearest Neighbor**
   1. *The psuedo-code that was given:*

NearestNeighbor(P)

p=p0

i=0

While there are still unvisited points

i=i+1

Select pito be closest unvisited point to pi-1

Visit pi

Return to p0 from pn-1

* 1. *Explanation of my code:*

First, I create a boolean array called seen. I give it a for a loop that increments up to the total number of points given (n). This allows us to check later in our while loop if we have visited a point or not; if the result is true, then we move on to the next point. This continues until it returns true for all available points.

Instead of a while loop, I created a for loop that increments up to the total number of points (n). First it calculates the distance between the primary point and between the current incremented point. If the calculated distance is shorter than the previous distance, it is replaced/stored for the value of distance. I then store the point that has the shortest distance into a new array that contains the final results. I then increment to the next point and repeat the process until the shortest distances and their respective points are saved in incrementing order.

1. **Exhaustive Search**
   1. *The psuedo-code given:*

OptimalTSP(P)

d=∞

For each use of the n! permutation Pi of point set P

If(cost(Pi) ≦ d) then d=cost(Pi) and Pmin = Pi

Return Pmin

* 1. *Explanation of my code:*

First, I create an array that stores the same value of its corresponding index. (index 0 will contain value 0, index 1 will contain value 1).This will allow the next\_permutation function to create the next possible permutation of the array. The do-while loop is representative of O(n!), within it we calculate the distances between points. If the distances between points is shorter than the current shortest distance, we save that distance into our permIndex array and the rest of the values in the array are shifted up, thus giving us shortest distance in descending order. Once that has been calculated, distance has been reset and the do-while loop makes next\_permutation run and we move on to the next possible set of permutations. This continues until all possible set of permutations is complete.

**2. [15 pts] Determine the worst-case time complexity of your algorithms in terms of n. (This**

**will depend on your implementation.)**

* 1. Nearest Neighbor
     1. O(n2)
     2. There are two for loops, each is initiated n times, thererfore I have determined that the worst-case time complexity is n2.
     3. for(int k=0; k < n; k++)

{

int index=0;

distance=100000;

tempx=finalX[k];

tempy=finalY[k];

for(int j=0; j < n; j++)

{

d = sqrt((pow((tempx - xArray[j]),2)+(pow((tempy-yArray[j]),2))));

if( d < distance && seen[j] == 0)

{

distance = d;

index=j;

}

}

seen[index] = 1;

finalX[k+1] = xArray[index];

finalY[k+1] = yArray[index];

}

* 1. Exhaustive Search
     1. O(n! \* n)
     2. The do-while loops has n! permutations and thus has n! time. The for loop calculating the distance is ran through approximately n times. This results in n! \* n.
     3. do{

for(int i=0; i < (n-1); i++)

{

distance=distance + sqrt(pow((xArray[permIndex[i+1]]-xArray[permIndex[i]]),2)+(pow((yArray[permIndex[i+1]]-yArray[permIndex[i]]),2)));

//cout << endl;

//cout << "Current distance (total so far) is: " << distance << endl;

//cout << "Current point is: " << xArray[permIndex[i]] << "," << yArray[permIndex[i]] << endl;

}

//cout << "Current distance (final total) is: " << distance << endl;

if( distance < d || d == 0.0)

{

d=distance;

for(int saveShort=0; saveShort < n; saveShort++)

{

shortestRoute[saveShort]=permIndex[saveShort];

}

}

distance = 0.0;

}while(next\_permutation(permIndex,permIndex+n));

**3. [20 pts] Use a random number generator to devise inputs for your algorithms for at least**

**four different values of n. The values of n may need to be different for the two approaches**

**and should be chosen with the following in mind:**

**(a) n should be large enough so that you can reliably determine the runtime of your algorithm by using an appropriate timing function call such as clock() to time your program; i.e.,**

**the minimum run time should be at least 10 times more than the resolution of your clock**

**function (the smallest unit of time that it measures).**

**(b) Also choose n so that you can experimentally verify the theoretical runtime you derived above.**

**For each n, determine the run time by taking the average of three runs on the same input.**

**This reduces the likelihood of inaccuracies due to system load. Display your results in a table.**

**Explain your choice of n.**

----------------------------------------------------------------------------------------------------------------------------

**4. [10 pts] Match theory and practice: Argue/demonstrate that your experimental runtimes are**

**consistent with the theoretical complexities you derived.\***

I chose n for Exhaustive Search to be 7-10, in order to ensure that the runtimes given would not just be printed out as 0. Anything smaller gives us 0 and anything greater takes too long to run. Calculation based on answer in Question 2.

I chose n for NearestNeighbor by doing increments of 500 starting from 1000. This will allow for a truly good runtime. Small numbers in Nearest Neighbor tend to be within the 0 microsecond range, so in order to ensure that something would be printed during the runtime, I chose big numbers. Calculation based on answer in Question 2.

Below is a table of the four samples, their points, and their runtimes (program generated and calculated):

***Exhaustive Search:***

|  |  |  |
| --- | --- | --- |
| **n** | **Program generated** | **\*Calculated** |
| 7 | .01 microseconds  .008 microseconds  .01 microseconds  **Average:**  .00933 microseconds | 7 is our base number to compare the ratios with. |
| 8 | .063 microseconds  .065 microseconds  .064 microseconds  **Average:**  .064 microseconds | (7 \* 7!)/(8 \* 8!) = .109375  (.00933)/(.064) =.14578125  May be off due to other noise running in the background. |
| 9 | 059 microseconds  0.624 microseconds  0.58 microseconds  **Average:**  .598 microseconds | (7 \* 7!)/(9 \* 9!) =.0108  (.0933)/(.598)=.0156  May be off due to other noise running in the background. |
| 10 | 6.984 microseconds  6.634 microseconds  6.312 microseconds  **Average:**  6.64333 microseconds | (7\*7!)/(10 \* 10!)= .000972  (.0933)/(6.6433)=.014044  May be off due to other noise running in the background. |

***Nearest Neighbor:***

|  |  |  |
| --- | --- | --- |
| **n** | **Average** | **\*Calculated** |
| 1000 | .13 microseconds  .124 microseconds  .126 microseconds  **Average:**  0.12667 microseconds | 1000 is our base number to compare ratios with |
| 1500 | 0.275 microseconds  0.268 microseconds  0.269 microseconds  **Average:**  0.268 microseconds | (10002)/(15002)=0.444  (.12667)/(.268)=0.472  May be off due to other noise running in the background. |
| 2000 | 0.484 microseconds  0.48 microseconds  0.482 microseconds  **Average:**  0.482 microseconds | (10002)/(20002)=0.25  (.12667)/(.482)=0.262  May be off due to other noise running in the background. |
| 2500 | 0.765 microseconds  0.745 microseconds  0.747 microseconds  **Average:**  0.76533 microseconds | (10002)/(25002)=0.16  (.12667)/(.76533)=0.165  May be off due to other noise running in the background. |