Cheetah Group: Steven Dea, Jenny Jeon, Frances Soman
EP.605.620 Algorithms for Bioinformatics
Spring 2020 Dr. Rubey

Group Project # 2: Solution for the Bitonic Euclidean Traveling Salesman Problem

Introduction

The traveling salesman problem has its roots in recorded mathematical puzzles dating back three centuries, and related even to the Knight's tour puzzle, notable in its earliest recorded reference by 9 century poet Rudrata (*Kavyalankara,* Rudrata, c900), in which the knight visits every square on the chess board once. If the knight ends on one move away from the beginning square, it is a closed path. In 1736, mathematician Leonhard Euler proposed the problem of the Seven Bridges of Koenigsberg (Euler, 1736), to find a unidirectional path around the city crossing each bridge only once, ending at the starting point. Later versions of this type of puzzle was proposed by W.R. Hamilton (c1856) to find a unidirected graph which visits each vertex once also starting and ending with the same vertex, to which Hamilton ascribed the term "Hamiltonian cycle". This problem has been labeled the Traveling Salesman Problem and has been studied through the centuries due to historical foundation for graph theory, its general applicability to practical problems and, more importantly, due to the difficulty of its solution.

Many heuristics and methods have been applied to solve this important problem in graph theory. The method suggested to use in this solution is applying dynamic programming to break up the problem, and apply the principles of the bitonic tour. The term bitonic refers to a path through vertices in which each edge is visited only once, with no edge crossing the path of any other. The optimal bitonic tour provides the path of minimum distance. The solution requested for this problem requests a Euclidean distance with respect to an x-y coordinate plane. In this proposed solution, a set of points with x-y coordinates is sorted by x, and the edges of minimum Euclidean distance between vertices starting from the leftmost vertex (based on the smallest value of x in the sorted array of points) to the rightmost vertex is determined within the principles of a bitonic tour of minimum distances. The path then is traversed from the rightmost vertex to the left, with the vertices not included in the first tour path. This should result in the optimal solution for the entire cycle from the first vertex, ending at the first vertex with no edges crossing paths and providing the minimum cycle distance.

Application to Bioinformatics

The Traveling Salesman Problem (TSP) in graph theory has many applications in bioinformatics analysis of biomedical research. Applications using the TSP or Hamiltonian cycle in this field vary whereas some use algorithms analyzing behavioral characteristics of biological groups, such as using the Dynamic Flying Ant Colony Optimization algorithm, for finding optimal solutions in this group of NP-complete problems (Dahan et al, 2019).

An interesting application of using the TSP in analysis in bioinformatics is in formulating analysis of evolutionary genetic codon distances as a TSP problem (Attie et al, 2019). To test the theory of evolutionary robustness and connectivity of the standard genetic code (SGC) through local or global optimization at the translational level (specifically with codon distances) and at the amino acid level, the TSP was used to model the SGC. A Hopfield neural network to represent the evolutionary genetic adaptive behavior was used to solve this TSP model. Another example of using the TSP in analyzing biomedical data is modeling correlation of expression changes in genes involved in Alzheimer's disease progression (Cruz-Rivera et al, 2019). The TSP model was used to identify cyclical paths in correlation of gene expression changes in the progression of neurodegenerative diseases and Alzheimer's. The results identified several genes of known correlation in neurodegenerative diseases and in Alzheimers, and identified novel candidate genes which show potential correlation in these diseases.

A particularly unique and novel use of bioengineering to solve the TSP lies not in using behavioral algorithms but using biological cells to serve as vehicles for computational machines in solving the TSP (Esau et al, 2014). As a biological parallel computing mechanism, E. coli cells were genetically programmed with marker gene plasmids as a way to model the TSP. The plasmids create phenotype selective paths in which the optimal route was defined by the frequency occurrence of particular phenotypes. A four destination TSP was modeled with specific plasmids with specific phenotypes that can be selected, such as ampicillin resistance and color. Paths were designed using plasmid linkages. Distance correlation was designed with respect to plasmid ligation and gene concentration, with a biological bias toward shortest paths. The results showed an accurate modeling of a four destination TSP. The concept of using biomolecules as rapid and expansive parallel mechanisms for solving optimization problems such as the seven point Hamiltonian Path problem was first used by L.M. Adleman in 1994 using DNA amplification (ref in Esau et al, 2014).

Efficiency of Proposed Solution

Analysis of efficiency for the proposed solution:

The proposed solution contains one set of embedded for loops in the path distance analysis, with For i from i from 0 to n, the For j from 1 to n followed by For k = 1 to n (lines 9-23). This has an efficiency upper bound of O(n^2). There are three separate For loops (lines 24 – 30) for adding the unused vertices to the rightArray, and assembling the leftArray and rightArray into the final solution array. All of these three For loops together have an efficiency upper bound of O(3n).

Analysis of efficiency for the MERGE-SORT:

The MERGE-SORT was chosen as a time efficient solution for sorting vertex points with their respective coordinates for the TSP path analysis. The efficiency of O(nlgn) is one of the best upper bound execution times for a sorting algorithm and will efficiently handle larger sets of vertex points and coordinates.

Pseudocode

//Create a class P that has two member variables, an x-value, and a y-value.

```
Class P
    x
    y
```

//Sort all of the P objects according to their x-values using mergeSort and insert it into a new array, sortedArray of size n. [O(nlogn)]

```
SortedArray = MERGE-SORT(P,0,n)
//*By mergesort:
MERGE-SORT(A, p, r)
    if p < r
        q = (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
MERGE(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1.. n1 + 1] and R[1 …n2 + 1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i – 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1 +1] = infinity
    R[n2 + 1] = inifinity
    i = 1
    j = 1
    for k = p to r
        if L[i] <= R[j]
            A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
```

//Create a method to solve for the distance between two P objects. It will take in the two P objects in question and get the x and y values for each of them. It then solves for distance (sqrt[(x2-x1)^2 +(y2-y1)^2]) and then returns this distance.

```
DISTANCE(p1, p2)
    int x1 = p1.getX()
    int y1 = p1.getY()
    int y2 = p2.getY()
    int distance = sqrt[(x2-x1)^2 +(y2-y1)^2]
    return distance
EUCLIDEAN-TSP(sortedArray, n)
    leftArray[n]
    leftArray[0] = sortedArray[0]
    leftArray[] = [p1]
    rightArray[n]
    rightArray[] = []
    distanceArray[n]
    unusedArray[n]
// For leftArray
    k = 1
    for i = 0 to n:
// Reset distance array to an empty array of size n-i
        distanceArray[n-i]
        for j = 1 to n:
            d = distance(sortedArray[i], sortedArray[j])
            distanceArray[j] = d
        min_d = min(distanceArray)
        leftArray[i] = min_d
// Set the new i and j to the index of the selected value
        i = index(min_d) + 1
        j = index(min_d) + 1
// Add "skipped" P objects to unusedArray
        for k to n:
            if i > k:
                unusedArray.append(sortedArray[k])
            if k == i:
                k += 1
                break
// For rightArray
    for i = 0 to unusedArray.length():
// Add to rightArray by going right-to-left from unusedArray
        rightArray[i] = unusedArray[unusedArray.length()-i]
// Merge leftArray and rightArray into solution
    solution = [n+1]
    for i = 0 to leftArray.length():
        solution[i] = leftArray[i]
    for j = 0 to rightArray.length():
        solution[j+leftArray.length()] = rightArray[j]
// return the solution array back to the left-most-point
    solution[n] = leftArray[0]
    return solution
```

I/O Examples

Example 1

Input	Intermediate	Output

Dataset:
p1 = (1,7)
p2 = (2,1)
p3 = (3,4)
p4 = (6,5)
p5 = (7,2)
p6 = (8,6)
p7 = (9,3)
 	
Initialized Variables:
n=7
leftArray[n]
rightArray[n]
distanceArray[n]
unusedArray[n]
 	
leftArray = sortedArray[0]
rightArray = []
	Call mergeSort(DataSet) to sort all P objects according to their corresponding x-values and place these values into sortedArray.
	
	For each current P, compare the distances for every point to the "right" (higher x-value) and find the next P-object with the closest distance to add into leftArray.
	
	For every value less than our selected value that is not selected, we check it against our value of k to determine if it has been "skipped" and add it to unusedArray if yes.
	
	Iterate through unusedArray from the highest index to lowest index and add it to rightArray.
	
	Merge leftArray and rightArray by iterating up leftArray and rightArray and adding a final P of sortedArray[0] to the end to show a full loop.
		sortedArray = [p1, p2, p3, p4, p5, p6, p7]
		
		leftArray = [p1, p3, p4, p6, p7]
		
		unusedArray = [p2, p5]
		
		rightArray = [p5, p2]
		
		solution = [p1, p3, p4, p6, p7, p5, p2, p1]

Example 2

Input	Intermediate	Output
Dataset: p1 = (1,7) p2 = (3, 6) n=2 leftArray[n] rightArray[n] distanceArray[n] unusedArray[n]	Call mergeSort(DataSet) to sort all P objects according to their corresponding x-values and place these values into sortedArray. For each current P, compare the distances for every point to the "right" (higher x-value) and find the next P-object with the closest distance to add into leftArray. For every value less than our selected value that is not selected, we check it against our value of k to determine if it has been "skipped" and add it to unusedArray if yes. Iterate through unusedArray from the highest index to lowest index and add it to rightArray. Merge leftArray and rightArray by iterating up leftArray and rightArray and adding a final P of sortedArray[0] to the end to show a full loop.	**sortedArray** = [p1, p2] **leftArray** = [p1, p2] **unusedArray** = [] **rightArray** = [] **solution** = [p1, p2, p1]

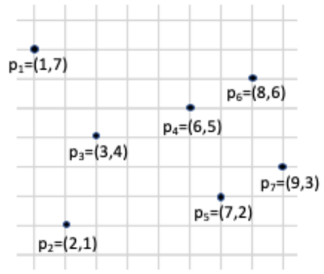
Example 3

Input	Intermediate	Output

Dataset:
p1 = (1,1)
p2 = (2, 3)
p3 = (3, 2)
p4 = (4, 7)
p5 = (5, 3)
 	
n=5
leftArray[n]
rightArray[n]
distanceArray[n]
unusedArray[n]
	Call mergeSort(DataSet) to sort all P objects according to their corresponding x-values and place these values into sortedArray.
	
	For each current P, compare the distances for every point to the "right" (higher x-value) and find the next P-object with the closest distance to add into leftArray.
	
	For every value less than our selected value that is not selected, we check it against our value of k to determine if it has been "skipped" and add it to unusedArray if yes.
	
	Iterate through unusedArray from the highest index to lowest index and add it to rightArray.
	
	Merge leftArray and rightArray by iterating up leftArray and rightArray and adding a final P of sortedArray[0] to the end to show a full loop.
		sortedArray = [p1, p2, p3, p4, p5]
		
		// Because there is a tie in distance between p2 and p3, p2 will be selected first to insert into leftArray and then p3 will be selected as the min in our next iteration
		leftArray = [p1, p2, p3, p5]
		
		unusedArray = [p4]
		
		rightArray = [p4]
		
		solution = [p1, p2, p3, p5, p4, p1]

Runtime Analysis

Using a coordination system, give each point to their own x, y coordinates and sort the points based on the coordinates.



Set 4 Array as the following: sortedArray,
leftArray,
rightArray,
distance Array, and
unusedArray

Assign the 7 point's elements in the sortedArray, p_1 in the leftArray, and leave empty the leftArray, distanceArray and unusedArray.

//For the leftArray T(n^2)
Start from sortedArray[i=0] and compare the distance among$|p_1 p_2|$ to $|p_1 p_7|$. T(n)

Each distance value, j, is stored in the distanceArray[].

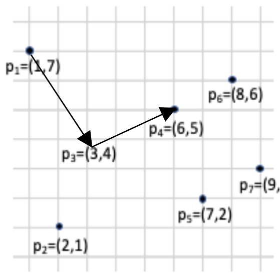
The point of shortest distance assigned by i, and add it into the leftArray[].

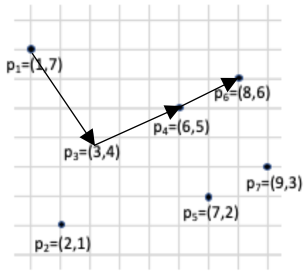
Move to sortedArray[index(min_d) + 1] and compare with k.

1. If sortedArray[index(min_d) + 1] > k,
   unusedArray.append(sortedArray[k])
   k += 1;
2. If sortedArray[index(min_d) +1] == k,
   k += 1;
3. If sortedArray[index(min_d) + 1] < k,
   compute and compare the distance.

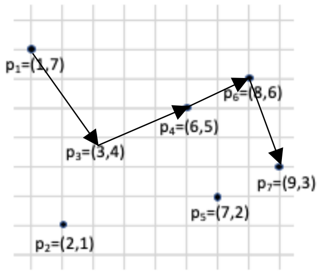
In this case sortedArray[i](= p_3) > k= p_2
So, p_2 add to unusedArray[].
Move to sortedArray[2].

In this case sortedArray[i](= p_3) = k (= p_3). So, compare the distance $|p_3\, p_4|$, $|p_3\, p_5|$, $|p_3\, p_6|$, $|p_3\, p_7|$.

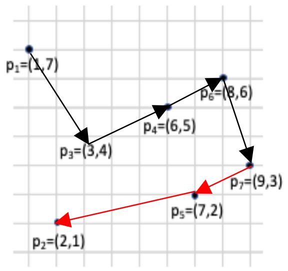


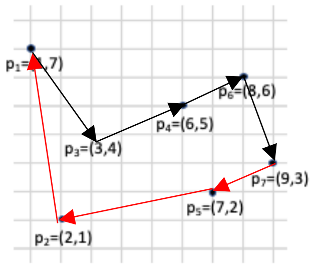


When there are no elements to compare, add the last index in sortedArray, p_7 into the unusedArray[] and terminate the for loop.



//For the leftArray T(n)
"going strictly **leftward**" means the indices of the sorted points form a
strictly **decreasing** sequence as well. Iterate through unusedArray from the highest index to lowest index and add it to rightArray.





Merge leftArray and rightArray and print soution[]. T(n)

Run time analysis for MERGE-SORT:

$$T(n) = \begin{cases} \theta(1) \\ aT(n/b) + D(n) + C(n) \end{cases}$$
$$= T(nlgn)$$

Run time analysis for our proposed solution:

$$T(n) = T(n^2) + 3T(n)$$
$$\sim= T(n^2)$$

Overall running time:

$$T(n) = T(nlgn) + T(n^2)$$

Reason: The nested for loop for i and j to solve for distances are n^2 time. There are 3 for loops after our nested for loop that contribute 3n cost time. Lastly, to print out the output takes constant time. Therefore, the overall time cost of our solution is MERGESORT O(nlgn) + EUCLIDEAN-TSP $O(n^2)$.

Correctness of Proposed Solution

Initialization:

For MERGE-SORT: (pp 53 – 54 in CLRS text)

The loop invariant for the MERGE section (lines 10 – 17) holds as prior to the first iteration as follows: at the beginning of the loop, k = p and the array A has not had the arrays L and R copied into it yet. Since I = j = 1, both L and R arrays are indexed to the smallest elements.

For the solution pseudocode:

At the beginning of the For loop, k, i and j are initialized and the distanceArray is set to an empty array, and the leftArray, rightArray and unusedArray are empty.

Maintenance:

For MERGE-SORT:

At each iteration of the loop invariant (lines 10 – 17), either the L array is copied into array A and i is incremented or the R array is copied into array A and j is incremented. k in the for loop is incremented.

For the solution pseudocode:

At each iteration in the For loops, I, j, and k are incremented. In the first set of for loops, the minimum distance is added to the leftArray and I and j are set to the index of the minimum distance element. In the unusedArray for loop, k is incremented with each iteration and the unusedArray is appended with the kth element of the sortedArray. In the rightArray for loop, i is incremented with each iteration and the rightArray is given the unusedArray element at unusedArray.length – i.

Termination:

For MERGE-SORT:

At the termination, k = r + 1, A contains all the elements in L and in R, in sorted order, which is $n_1 + n_2 + 2 = r - p + 3$ elements. The sentinels (lines 8-9) are the only elements not copied back into A.

For the solution pseudocode:

At termination, in the for loop for the leftArray, i and j are = n and the leftArray contains the minimum distance elements from sortedArray. In the for loop for the unusedArray, k = n or k = i and and the unusedArray contains the elements from sortedArray skipped in the leftArray loop. In the for loop for the rightArray, I = unusedArray.length and the rightArray contains the unusedArray elements.

References

Attie O., Sulkow B., Di C., Qiu W., 2019, “Genetic codes optimized as a traveling salesman problem”, *PLoS ONE,*, 14(10)

Cruz-Rivera Y., Perez-Morales J., Santiago Y., Gonzalez V., Morales L., Cabrera-Rios M., Isaza C., 2018, “A Selection of Important Genes and Their Correlated Behavior in Alzheimer’s Disease”, *J Alzheimer’s Disease*, 65, 193-205

Dahan F., El Hindi K., Mathkour H., AlSalman H., 2019, “Dynamic Flying Ant Colony Optimization (DFACO) for Solving the Traveling Salesman Problem”, *Sensors(Basel)*, 19(8), 1837

Esau M., Rozema M., Zhang T.H., Zeng D., Chiu S., Kwan R., Moorhouse C., Murray C., Tseng N., Ridgway D., Sauvageau D., Ellison M., 2014, “Solving a Four-Destination Traveling Salesman Problem Using Escherichia coli Cells as Biocomputers”, *ACS Synthetic Biology,* 3, 972-975

Euler, Leonhard (1736). "Solutio problematis ad geometriam situs pertinentis". *Comment. Acad. Sci. U. Petrop* 8, 128–40 (Wikipedia)

Hamilton, William Rowan, (1956) “Memorandum respecting a new system of roots of unity”, *Philosophical Magazine,* 12, 446 (Wikipedia)