HOMEWORK 4

Name: Tianyu Gu Collaborator: AngLi

Q1:

The problem can be solved by dynamic programming:

For the node, in order to find if there is a path from to with k edges, we could check all the paths from to, if exists. For example, if there is a path from to , this problem becomes if there is a path from to with exactly k-1 edges. This algorithm will run in O(n!) times at the worst case. Therefore, we need to memorize all the internal computations, by which we could decrease the computation dramatically.

Pseudo code:

1. Matrix\_Opt[N][k] = Nil  // declare a two dimensional array
2. OPT(G,k):               // the function used to find the path. Suppose the node in G is arranges from V1  to Vn //
3. **if**(k > n):
4. **return** 0        // Apparently,there is no path when k > n.
5. **if**(k == 0 and Vn != V1):
6. **return** 0        // this means it cannot reach V1 with k edges.
7. **if**(k == 0 and Vn == v1):
8. **return** Vn       // this means a valid path has been found.
10. // the above are base cases//
11. **else**:
12. **while** the path (Vi,Vn) exists:          //search all the paths ends with Vn. i is range(1,n)
13. **if** Matrix\_Opt[Vi][k-1] != Nil:
14. temp = Matrix\_opt[Vi][k-1]
15. **else**:
16. G' = G with only V1 to Vi       //cut the graph and keep the node V1 to Vi, since in this line-graph, there is no path backward.
17. temp = OPT(G',k-1)              // the problem has become a subproblem with smaller v and  k.
18. Matrix\_Opt[n][k] =temp          // record the OPT(Gn,k), since this is the first time of
19. Computing this.
20. **if** temp != 0:                   // temp != 0 means that we found a path
21. Vn.π = Vi                   // this record the actual path by recording the last node in the
22. Path.
23. **return** Vn                   // return the node we found
25. end **while**
26. **return** 0                                // since we did not find any path from Vi to Vn with edges k, then return 0
27. // which means the path with k edges cannot be found.

**Correctness:**

Let Path (, k) gives the path from to with edges k, otherwise Nil. we could find that, for any points in this line-graph, let’s say. If there is a path from towith k edges, we can know that it could be expressed as:

that edge (.

Therefore, the existence of really depends on if there is ani makes true.

**Runtime analysis:**

In the worse case, the number of edges k will equal to n.

Therefore, in this case, in order to solve the, we need to solve sub problems, ranging from to. Since we use a memorization method, which enables the algorithm only solve each sub problem once. For each problem, in the worse case, we need n times of iteration to solve (line 12 ~ line 25). That is to say: for solving we need 1 unit of computation, for and we need 2 unit of computation each.

Therefore, the total running time is bounded by:

This running time, which can be proved inductively, equals to O().

Q2.

(a).

We can use the similar algorithm discussed in the class, with several small changes:

* 1. Delete the constant trade-off factor ƛ.
  2. Add a new factor k, to count the numbers of partitions.

Therefore, for the points, and k segments, the problems can be reduced by the rules:

Therefore, we could establish the whole OPT table increasingly by iteration.

**Pseudo code:**

Function (j, k):

Array =Nil

Set

For all pairs

Compute the least squares error for the segment

End for

// the iteration used to compute the error i j//

For j = 1, 2…n:

For i=1,2,..k:

Use the recurrence above to compute

Endfor

Endfor

Return OPT (n,k)

// the iteration used to update the table opt//

// the function used to find the exacted segments//

Find-Segments (n, k, opt (n, k)):

If n=0 or k=0 then:

Output nothing

Else:

Find an i that minimize

Return the i and Find-segments(i-1,k-1)

(b).

The program below has been implemented by java.

My answer for segmentation is: [1, 37, 59, 82, 100], and the error is 71.11648043771159.

There is a slightly difference between the real implementation and the pseudo code, in the program, I use a 2d matrix path to keep tracking i that minimize for opt (n, k), so that the find-segments function is not necessary now.

It’s a trade-off between space efficiency and time efficiency, since the problem only give n=100, it will not introduce any problems.

Code:

**import** java.io.FileNotFoundException;  
**import** java.io.FileReader;  
**import** java.util.Scanner;  
  
**import static** java.lang.Math.*pow*; *//only used to compute power***public class** hw4\_2{  
 *// the function used to read in the files hw4test.txt.* **public static double**[][] readfile() **throws** FileNotFoundException {  
 **double**[][] vertices= **new double**[100][2];  
 Scanner in = **new** Scanner(**new** FileReader(**"hw4test.txt"**));  
 **int** n = 0;  
 **double** temp\_x,temp\_y;  
 **while**(in.hasNextLine()){  
 temp\_x= in.nextDouble();  
 temp\_y = in.nextDouble();  
 vertices[n][0] =temp\_x;  
 vertices[n][1] = temp\_y;  
 n++;  
 }  
 **return** vertices;  
  
 }  
 *//this function is used to compute the a in linear regression for points from m to n.* **public static double** compute\_a(**int** m,**int** n,**double**[][] vertices){  
 **double** sum\_x=0;  
 **double** sum\_y=0;  
 **double** sum\_xy=0;  
 **double** sum\_x2=0;  
 **double** number = n-m+1;  
 **for**(**int** i=m-1;i<n;i++){  
 sum\_x = sum\_x + vertices[i][0];  
 sum\_y = sum\_y + vertices[i][1];  
 sum\_xy = sum\_xy + vertices[i][0] \* vertices[i][1];  
 sum\_x2 = sum\_x2 + *pow*(vertices[i][0],2);  
 }  
 **double** numerator = number \* sum\_xy - sum\_x \* sum\_y;  
 **double** denominator = number \* sum\_x2 - *pow*(sum\_x,2);  
 **return** numerator/denominator;  
  
 }  
 *//this function is used to compute the b in linear regression for points from m to n.* **public static double** compute\_b(**int** m,**int** n,**double**[][] vertices){  
 **double** sum\_x=0;  
 **double** sum\_y=0;  
 **double** number = n-m+1;  
 **double** a = *compute\_a*(m,n,vertices);  
 **for**(**int** i=m-1;i<n;i++){  
 sum\_x = sum\_x + vertices[i][0];  
 sum\_y = sum\_y + vertices[i][1];  
 }  
 **double** numerator = sum\_y - a \* sum\_x;  
 **return** numerator/number;  
 }  
 *//this function is used to compute the error r for points m to n.* **public static double** compute\_r(**int** m,**int** n,**double**[][] vertices){  
 **double** error = 0;  
 **double** difference = 0;  
 **double** a = *compute\_a*(m,n,vertices);  
 **double** b = *compute\_b*(m,n,vertices);  
 **if**(m==n) {**return** 0;}  
 **for** (**int** i = m-1;i<n;i++){  
 difference = vertices[i][1]-a\*vertices[i][0]-b;  
 error = error + *pow*(difference,2);  
 }  
 **return** error;  
 }  
 *//this function is used to calculate the OPT tables* **public static double** OPT(**int** n,**int** k,**double**[][] vertices) {  
 **double**[][] Matrix\_Opt = **new double**[n+1][k];  
 **double** error[][] = **new double**[n+1][n+1];  
 **double** temp = 0;  
 **double** min = 1000; *// a number to assign min to infinite, so we can update it.* **int**[][] path = **new int**[n+1][k]; *//used to keep tracking the points to segment.* **int** previous = 0;  
 **int**[] trace = **new int**[k];*// a small matrix used to print the segmentation* **int** num = k;  
 *//compute the error matrix Eij for all i and j* **for** (**int** i = 1; i < 100; i++) {  
 **for** (**int** j = i; j <= 100; j++) {  
 error[i][j] = *compute\_r*(i, j, vertices);  
 }  
 }  
 *// iteratively compute the OPT matrix* **for** (**int** j = 1; j <= 100; j++) {  
 **for** (**int** i = 0; i < k; i++) {  
 **if** (j == 1 || j == 2) { *//initialize the value* Matrix\_Opt[j][i] = 0;  
 }  
 **else** {  
 *// when k=0, it means only one line left, the OPT value is the same with error value.* **if** (i == 0) {  
 Matrix\_Opt[j][i] = error[1][j];  
 }  
 **else** {  
 *// follow the dynamic rules to update the table* **for** (**int** m = 1; m < j; m++) {  
 temp = Matrix\_Opt[m-1][i - 1] + error[m][j];  
 **if** (temp < min) {  
 min = temp;  
 previous = m;  
 }  
 }  
 Matrix\_Opt[j][i] = min;  
 path[j][i] = previous;  
 min=1000;  
 }  
 }  
 }  
 }  
 *//print out the exact segmentation points* trace[num - 1] = path[n][num - 1];  
 **while**(num>1) {  
 System.***out***.println(trace[num - 1]);  
 trace[num - 2] = path[trace[num - 1]][num - 2];  
 num--;  
 }  
 **return** Matrix\_Opt[n][k-1];  
  
 }  
 **public static void** main(String[] args) **throws** FileNotFoundException {  
 **double**[][] vertices;  
 vertices = *readfile*();  
 **double** temp = *OPT*(100,4,vertices);  
 System.***out***.println(temp);  
  
 }  
}

Q3:

First, according to the question, we can safely assume the total number of precincts n is even number.

Secondly, we can assume the total registered voters for party A is larger than B. therefore, we can only focus on party A, since it’s impossible for party B to have majority in both district.

Notations:

Use to represent the precincts.

Use to represent the number of registered voters for party A in each precinct.

Use m to denote the total number of registered voters in each precinct.

The total voters for A will be denoted as and we know

Use q to represent the total number of precincts in one district.

Use to denote the sum of which belongs to the district of s with q numbers of precincts.

**Algorithm:**

The problem can be solved by dynamic programming, the function gives whether for, there exists a district with q precincts and voters.

Therefore, we could know that:

That means, in order to find a district with m precincts and voters in, it has to be in one of the following situations:

* 1. We can find a district with q precincts and voters in.
  2. We can find a district with q-1 precincts and voters in, so that we can put in this district.

The base case is simple:

is True.

Therefore we can build the table iteratively, based on the recursion rules above

for all.

**Solution:**

We know that, in one district, if voters for party A want to dominate, the in this area must larger than nm/4. Moreover, since A need dominate in both districts, so that also need to be larger than nm/4.

Therefore, after we compute the whole OPT table, we could check all the values among where . Then we can know that, it is susceptible for gerrymandering if any of these values is true.

**Runtime Analysis:**

For computing each entry in this OPT table, only constant time is needed, since you compute it increasingly using iteration. Therefore, the whole running time is bounded by number of entries, which is