B11901095 電機二 朱振瑀 Algorithm PA2 Report

1. Defining Subproblems

maximumPlanarSubset(i,j) denotes the maximum number of non-overlapping chords on the circle from vertex i to j. To solve the main problem, we want to get the value of maximumPlanarSubset(0,2n-1), where 2n is the number of vertices. We require that j > i, or the function would just return 0.

2. Data Structures Used

a. chord storage

I use a 1D vector<int> chordList of size 2n to store all the chords. chordList[i] = j iff there is a chord connecting vertex i and j.(So if chordList[i] = j, chordList[j] = i). If a vertex i has no chords, then chordList[i] = -1.

b. dynamic table

I use a 2D vector<vector<int>> M to store the result of the subproblems. The table is designed to be triangular in order to enhance space efficiency. M has 2n rows, and each row M[i] is of size (2n-i), with i from 0 to 2n-1. The answer to the subproblem maximumPlanarSubset(i,j) is stored in M[i][j-i]. I also defined a function m(i,j) to directly return the value of M[i][j-i] for convenience.

3. Recurrsion Formulae

mps(i,j) =

0, if i <= j -case 1

1 + mps(i+1,j-1), if there exists a chord (i,j) -case 2

max(1 + mps(i,k-1) + mps(k+1,j-1), mps(i,j-1)),
if there exists k such that k is between i, j and there exists a chord (k,j).
-case 3

mps(i,j-1), otherwise -case 4

explaination and proof:

case 1:

In the definition of the subproblem, j > i is required. if i = j, it would return 0 since there are no chords whose start point is equal to end point.
i > j is redundant since “from vertex i to j” and “from vertex j to i” are equivalent, so we only need to calculate cases where i < j.

case 2:

If chord (i,j) exists, then it would not overlap with any of the chords lying entirely between i and j. Thus, that chord must be included in the solution (sets of chords) to the subproblem mst(i,j), and mst(i+1,j-1) is the subproblem that is contained in the range i to j but doesn’t consider vertices i and j, so mst(i,j) = mst(i+1,j-1) + 1

proof by cut and paste: assume we can find an optimal set of chords mps’(i,j) that does not include chord(i,j), then mps’(i,j) = mps(i+1,j-1). However mps’(i,j) is not optimal because mps(i,j) = 1 + mps(i+1,j-1) > mps’(i,j) = mps(i+1,j-1). So chord (i,j) must be included.

case 3:

If there exists a chord one of whose endpoints is j, and the other is between i and j (denoted by k), then we have to decide whether or not to include it. 1 + mps(i,k-1) + mps(k+1,j-1) is the result of including the chord, 1 means the chord(k,j), and mps(i,k-1), mps(k+1,j-1) are the two subproblems separated by the chord (so all chords in the two subproblems would not overlap). mps(i,j-1) is the result of not including the chord, we decrement j by 1(because if vertex j is not used, then m(i,j-1) is basically m(i,j) without an unused vertex, they are equal). We choose the maximum of the two.

case 4:

It means that no chord is attached to vertex j. So we decrement j by 1.

4. Time Complexity Analysis

a. reading input data

the process goes through each chord once, so Θ(chord). 2n endpoints can at most form n chord, so Θ(n).

b. initializing dynamic table(init() function)

the table is of size $(2n)^2/2$, and it sets each element in the table to -1, so $\Theta(n^2)$.

c. maximumPlanarSubset() function

this recursive function would solve each subproblem at most once, and there are $\Theta(n^2)$ subproblems, the operations in a subproblem all takes constant time, so $O(n^2)$.

d. output results

I use recursive output. although there are two parameters, i and j, they will get closer and closer, so $\Theta(n)$

overall time complexity is $\Theta(n^2)$

5. Space Complexity Analysis

it only uses a 1D vector chordList($\Theta(n)$), and 2D vector M($\Theta(n^2)$), so overall space complexity is $\Theta(n^2)$.

6. Result Verification

If the result satisfy the following properties, then it is correct:

a. number of chords in the MPS is correct
b. the number of output chords are equal to the first line of the output
c. all output chords are in chords from the input
d. no output chords overlap, and no repeated chords
e. output chords are sorted in increasing order in terms of first endpoint

proof of a.

To prove that a. stands, we have to prove the optimal substructure of the recursion formulae, which is done above.

proof of b.

In the output function, if j <= i, nothing is printed out, and mps(i,j) = 0, so it is correct.

if there is a chord (i,j), then the number of chords printed out = 1 + output(i+1,j-1). from mps(i,j) = 1 + mps(i+1,j-1), it is correct.

if there is a chord (k,j), then if mps(i,j) = 1 + mps(i,k-1) + mps(k+1,j-1), chord(k,j) is included, so number of chords printed out = 1 + output(i,k-1) + output(k+1,j-1). which is correct. if the equation does not hold or j has no attached chords, then number of chords printed out = output(i,j-1). from mps(i,j) = mps(i,j-1) in this case, it s correct.

proof of c.

When outputting chords, I use the chordList array. chord(i,j) is printed out only if chordList[i] = j, and, chord(k,j) is printed out only if chordList[k] = j. All data from the chordList is directly from the input file and is not changed throughout the entire process. Hence, all the output chords are chords from the input file.

proof of d.

When outputting, chord(i,j) does not overlap with any chords whose start points and end points are between i+1 and j-1. chord (k,j) does not overlap with any chords whose start points and end points are both between i and k-1, or between k+1 and j-1.

Above are the only cases possible for a chord to be printed out, so no output chords overlap with each other. Repeated chords are also considered to overlap, since there are no overlapping chords, no chords repeat.

proof of e.

When outputting, the start point of chord(i,j) < the start point of any chords whose start points and end points are between i+1 and j-1. (min i+1). Since the former is printed out before executing the latter subproblem, the start points would be sorted in increasing order.

Besides, the start point of any chords whose start points and end points are both between i and k-1(max = k-1) < the start point of chord(k,j) < the start point of any chords whose start points and end points are both between k+1 and j-1(min = k+1). Since the left subproblem is executed before printing out the chord(k,j) before the right subproblem is executed, the start points would be sorted in increasing order.

Above are the only cases possible for a chord to be printed out, and we use the start point as the first point, so output chords are sorted in increasing order in terms of first point.

program to verify b,c,d,e:

For b, we just check the lines of the output file.

For c, we can generate a chordList, but this time chordList[i] = j iff i < j and there is a chord (i,j). We loop through the output list, check whether for each chord, chordList[first] = second.

For d, we can perform a “valid parentheses” check, where the first endpoint of a chord is the upper bracket, while the second is the lower. All chords are considered different brackets.

For e, we just loop through the output list once to see if it holds.

I have made a valid-check program, the script is in the src directory. This valid-check program assumes that the first line of the output(calculated number of chords in the maximum planar subset) is correct.

7. Bottom-Up Approach

In this program, I used the top-down approach to solve the maximum planar subset problem. But what about the bottom-up approach? I also tried implementing it(the source code has been commented in the mst.h file). The time and space complexity for both approaches is $\Theta(n^2)$, but the top-down approach is faster. For example, when running the case for 2n = 10000 with battery saver on and the laptop unplugged(), the top-down approach takes about 3100ms, while the bottom-up approach takes about 6800ms. This is because in this problem, the top-down approach does not have to solve all possible subproblems, or fill in the whole dynamic table, making it more efficient.