Algorithms Programming Assignment 2-Report

A. Introduction

The task is to compute the maximum planar subset of a set of chords in a circle, ensuring that no two chords overlap. The goal is to find the largest subset where all chords are planar, meaning they do not cross each other.

B. Data Structure Description

1. Chords Array

An integer array `chords[0:2N-1]` is used to store the endpoints connected by each chord. This allows efficient lookup of the corresponding endpoint for each position in the circle of chords.

2. Dynamic Programming Table

- A triangular-shaped 2D array `mps_table[i][j]` is used to store the size of the maximum planar subset of chords for intervals defined by endpoints `i` and `j`, where `0 <= i <= 2N - 1` and `i <= j <= 2N - 1`.
- `mps_table` is implemented as a single continuous 1D array:
- This approach improves memory access efficiency by maintaining a sequential access pattern, which can enhance cache performance.
- The triangular structure is managed by adjusting the starting address of each row's pointer, allowing customization of the range for the second index `j` and flexible handling of subproblem intervals.

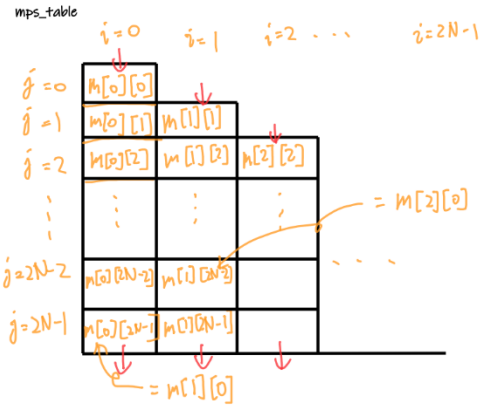

Figure 1 mps_table

All the `mps_table` is a continuous 1D array. That is, `m[0][2N-1]`, `m[1][1]` are connected, and `m[1][2N-1]`, `m[2][2]` are connected and so on. By making `m[1]` point to `m[0][2N-1]`, `m[2][0]` point to `m[1][2N-2]`…, we can get a triangle array `m[i][j]`, where `0<=i<=2N-1`, and `i<=j<=2N-1`.

3. MPS Array
   An integer array `mps[]` is used to store the extracted maximum planar subset of chords. This array holds the smaller indices of endpoints of the chords that form the maximum planar subset, as determined by the dynamic programming solution.

C. Algorithm

- The algorithm uses a **dynamic programming** approach to solve the problem efficiently.
- Recurrence relation:

$$M(i,j) = \begin{cases} 1 + \mathrm{M}(\mathrm{i}+1, \mathrm{j}\text{-}1), & \text{if chord}(\mathrm{i},\mathrm{j}) \text{ exists} \\ \max(\mathrm{M}(\mathrm{i}, \mathrm{j}\text{-}1), 1 + \mathrm{M}(\mathrm{i}, \mathrm{k}\text{-}1) + \mathrm{M}(\mathrm{k}+1, \mathrm{j})), & \text{if chord}(\mathrm{k},\mathrm{j}) \text{ exists and } \mathrm{i} < \mathrm{k} < \mathrm{j} \\ \mathrm{M}(\mathrm{i}, \mathrm{j}\text{-}1), & \text{otherwise} \end{cases}$$

- Initialization: The DP table is initialized with all zero values, as no chords are selected initially.
- Filling the table: For each pair of endpoints (i, j), the algorithm (int mps_len()) checks if there is a chord. Depending on whether it is an inner or outer case (if the chord crosses other chords), the table is updated based on the recurrence relation.
- Backtracking**:** Once the table is filled, the chords making up the maximum planar subset are extracted using a backtracking technique. The function get_mps(i, j) checks the chord(k,j) should be in the mps or not.

D. Time Complexity

- The table size is $2N*(2N+1)/2 = \Theta(N^2)$, then the process to initialize the table with 0 requires $\Theta(N^2)$. And the DP algorithm requires $O(N^2)$ time because we use top-down recurrence.
- The backtracking process: we can use l = (j-i) to measure T. In every recursive call, $T(l) <= \mathrm{Max}(T(l\text{-}2) + \Theta(1), T(l')+T(l\text{-}l'\text{-}2)+\Theta(1), T(l\text{-}1)+\Theta(1))$
  We can infer that the time for l decrease 1 is $\Theta(1)$. Then $T(\text{get\_mps}(0, 2N\text{-}1))= \Theta(N)$
- Overall, $T(N)= \Theta(N^2)+ O(N^2)+ \Theta(N)= \Theta(N^2)$