# Report of Maximum Planar Subset

Author: 陳郁安

Student ID: B10703104

## A. Basic Idea.

The Maximum Planar Subset problem can be solved easily using the concept of dynamic programming. To illustrate:

**a. Define The Problem.**

First we define the set of chords:

$\mathrm{C} = \{(\mathrm{i},\mathrm{j}) \mid \text{for any pair of i and j representing endpoints pair of the a chord}, \mathrm{i} < \mathrm{j}\}$

Define the problems:

$\mathrm{MPS}[\mathrm{i},\mathrm{j}]$ is the maximum planar subset in the chords set $\mathrm{C} \cap \{(\mathrm{x},\mathrm{y}) \mid \mathrm{x} \geq \mathrm{i}, \mathrm{y} \leq \mathrm{j}\}$

We want to solve $\mathrm{MPS}[0,2\mathrm{n}-1], \mathrm{n} = |\mathrm{C}|$

**b. Define Subproblems.**

To find $\mathrm{MPS}[\mathrm{i},\mathrm{j}]$, there are three conditions to consider.

<u>Condition 1. Endpoint j is connected to Endpoint k; and $\mathrm{i} < \mathrm{k} < \mathrm{j}$</u>

In this case, we have to decide whether do or do not count the chord (k, j) into the MPS, by considering if it makes the MPS larger. So we have:

$$\mathrm{MPS}[\mathrm{i},\mathrm{j}] = \begin{cases} \mathrm{MPS}[\mathrm{i},\mathrm{k}-1] \cup \mathrm{MPS}[\mathrm{k}+1,\mathrm{j}-1] \cup \{(\mathrm{k},\mathrm{j})\} \\ \mathrm{MPS}[\mathrm{i},\mathrm{j}-1] \end{cases}, \text{whichever has a larger size}$$

<u>Condition 2. Endpoint j is connected to Endpoint k; and $\mathrm{j} < \mathrm{k}$</u>

In this case, the chord (j, k) must not be in the MPS. So we have:

$$\mathrm{MPS}[\mathrm{i},\mathrm{j}] = \mathrm{MPS}[\mathrm{i},\mathrm{j}-1]$$

<u>Condition 3. Endpoint j is connected to Endpoint i</u>

In this case, the chord (i, j) must be independent to all the chords in $\mathrm{MPS}[\mathrm{i},\mathrm{j}-1]$. So:

$$\mathrm{MPS}[\mathrm{i},\mathrm{j}] = \mathrm{MPS}[\mathrm{i},\mathrm{j}-1] \cup \{(\mathrm{i},\mathrm{j})\}$$

The pattern is more clear and transparent now. To solve this problem, I decide to use bottom-up method, constructing MPS table from $\mathrm{j}-\mathrm{i} = 1$: 1 MPS[0,1], MPS[1,2], MPS[2,3]…, then $\mathrm{j}-\mathrm{i} = 2$: MPS[0,2], MPS[1,3], MPS[2,4]…. Lastly, $\mathrm{i}-\mathrm{j} = 2\mathrm{n}-1$: MPS[0, 2n-1], which is the answer.

## B. Programming for MPS.

### a. Recording the input data

Here I use fstream to read the input file and store the data using vector. For recording the endpoints data, in the beginning I was considering using a self-defined class called “pair_val”, but problem emerged when searching up an endpoint pair. For instance, we want to search a pair of value, with a given value 4, in a vector of “pair_val”:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 7 | 3 | 6 | 4 | 1 | 5 |
| 0 | | 1 | | 2 | | 3 | |

To search for the chord, we have to iterate from the first pair, and we have to check if 4 is equal to the first value or the second value. It’s complex and time costly (O(n)). In order to do “search” in O(1) time, I find out that we can instead store every single endpoint, the index of which is its corresponding endpoint. To illustrate, we modify from the vector above:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 0 | 7 | 6 | 1 | 4 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

The space complexity is same as above O(n) and searching or finding can be done in O(1), for every element and its index make an endpoint pair.

### b. Solving MPS and complexity analysis

Here I use bottom-up DP method, so there is some MPS table needs to be built; and, according to the 3-condition-subproblem that we mentioned, the pseudo code of building the MPS table can be illustrated as:

```
MPS_BUILD
for i = 0 to data.size() - 1
    MPS[i][i] = ∅
for gap = 1 to data.size() - 1
    for i = 0 to data.size() – gap - 1
        j = i + gap;
        c = data[j]
        MPS[i][j] = MPS[i][j-1]
        if (c < j && c >= i)    //otherwise it would be condition 2
            if (c == i)
                MPS = MPS[i][j]∪{(c,j)};    //condition 3
            else
                alt = MPS[i][c-1] ∪ MPS[c+1][j-1] ∪ {(c,j)}
                MPS[i][j] = alt if |MPS[i][j]|<|alt|    //condition 1
```

The pseudo code shows the concept. But in reality, the way we store the MPS set needs considering whether it is or is not efficient. It barely needs explaining that if we build a table of “set”, it would cost a lot of space. For example, we use a 2D vector representing MPS table, in which we stores vectors of chords; then, it would become a 3D vector. Although we can quickly get the size of the MPS and its element, the concept is too inefficient in terms of space. To improve this, I find that we don’t need to store the whole set, instead, we can just store “yes” or “no” to the question of “whether we need to let the chord (c,j) cut through M[i][j-1] to get M[i][j]” for the table of MPS, and we stores the size of MPS in another table.
In my program, I use a self-designed data structure called MPS_tool, the constructor of which builds the table of MPS and size of MPS:

```
MPS_Tool class
public member:
    MPS_Tool() // constructor
    Record_MPS() // pass an empty array, it would fill in the MPS
private member:
    MPS_SubsetCut // MPS table, stores bool data, “true” if (c,j)∈MPS[i][j]
    MPS_SetSize   // MPS size table, stores integer

MPS_Tool
for i = 0 to data.size() - 1
    MPS_SubsetCut[i][i] = false
for gap = 1 to data.size() - 1
    for i = 0 to data.size() – gap - 1
        j = i + gap;
        c = data[j]
        MPS_SubsetCut[i][j] = false
        MPS_SetSize[i][j] = MPS_SetSize[i][j-1]
        if (c < j && c >= i)    //otherwise it would be condition 2
            if (c == i)
                alt = MPS_SetSize [i][j-1] + 1
            else
                alt = MPS_SetSize [i][c-1] + MPS_SetSize [c+1][j-1] + 1
        if MPS_SetSize < alt
            MPS_SubsetCut[i][j] = true
            MPS_SetSize[i][j] = alt
```

The code is almost identical to the previous one, and it is more efficient than that

using 3D vector in terms of both space and time. If we use 3D vector to build the table, the the time used to fill in the table and space used are both:
$O(En^2)$, n is data size and E is the expected vector size of MPS
It is significantly larger than that of MPS_SubsetCut plus MPS_SetSize, which is $O(n^2)$ for both time and space.

Now considering the time used to record the MPS. If we use 3D vector method, we iterate every endpoints to record the chord set, which costs O(k), k is the size of MPS. Since we can use the built-in function size() for vector, the size of MPS costs O(1) to get. In conclusion, outputting both set and size costs O(k).

As for my program, I use `Record_MPS()` to record the set of MPS

```
Record_MPS(result, i, j)  //Record chords in MPS[i][j] into result
if (i >= j)
    return
k = MPS_SubsetCut[i][j]
if (!k)
    Record_MPS(result, i, j-1);
else
     c = data[j]
     result.push_back(c);
     Record_MPS(result, i, c-1);
     Record_MPS(result, c+1, j);
```

As illustrated, I only store the smaller endpoint of every chord in MPS[i][j], that is because we can find the other endpoint in O(1), so we can just deal with this when writing the output file:

```
fout << result.size() << '\n'         //write the MPS size, O(1)
for i = 0 to result.size()-1          //write the chords in MPS, O(k)
    fout << result[i] << " " << data[output[i]] << '\n'
```

The time complexity is O(k) as well. The method I use is absolutely more efficient than using 3D vector, whether in time or space.

## C.What I learn / My findings in this assignment.

1. To deal with paired value…

Storing paired values at the index corresponding to their pair is significantly more convenience than storing "pair" in a stack if "search" or "find" is frequently used, because it takes O(1) time which is faster than O(n) when data is big enough. However, the stack in which a value is paired with index can't be sorted, otherwise the pair-link between index and the value would mess up. So, if sorting is needed, and O(1)-time searching operation is also required, another stack storing the pairs is needed, so the unsortable one can be used for searching, and the sortable one can be used for sorting.

2. The MPS table and LCS table…

I find that the concept I use to construct the table of MPS (set) is similar to that of LCS in some sense. When the answer of subproblems is a set instead of a particular number. Rather than recording the whole set, it is easier to record the "direction" or "track", or the "how it changes" between the problem and the subproblem. By tracing the "track" from base problem to complex problem (LCS, tracing the direction from LCS[n-1,n] to LCS[0,n]) or otherwise (MPS, tracing the "how it changes" from MPS[0,n] to several subproblems), we can get the set of the complex in O(k), k is the amount of the "track".