2020 Fall Algorithms

B06102020 楊晴雯

Report on PA2: Maximum Planar Subset

1. Storing inputs

```
query_data[endpoint2] = endpoint1;
query_data[endpoint1] = endpoint2;
```

Swap the endpoints to keep endpoint1 always < endpoint2 for the convenience of backtracking. Use a *2n-1* vector to store the chords. For convenience, save both endpoints with their corresponding endpoints, so the search time for the other endpoint can be limited within Θ(1) by:

```
int k = query_data[j];
```

Cons: This will increase memory usage from *n* to *2n-1*. Although the space complexity does not change because of multiplication of a constant, doubling memory usage is certainly a burden to computers.

2. Dynamic Programming (max_planar_subset.cpp)

```
pair<int, vector<vector<unsigned short>>> Solution::Max_Planar_Subset()
```

I use Bottom up method, which requires to fill up two tables that contains all optimal substructures.

Recurrence:

$$M(i,j) = \max\left(M(i,j-1), M(i,k-1) + M(k+1,j) + 1\right) \ \ if\ k \in [i,j]$$
$$= M(i,j-1) \ \ if\ given\ a\ c_{kj}, k \notin [i,j]$$

To calculate the number of MPS, I declare a *(2n-1)*(2n-1)* table *Size* using *vector<vector<unsigned short>>*, because vectors don't require contiguous memory so it might work better than static matrix.

```
vector<vector<unsigned short>> Size(2 * n - 1, vector<unsigned short>(2*n-1, 0));

vector<vector<unsigned short>> Cases(2 * n - 1, vector<unsigned short>(2*n-1, 0));
```

The answer is stored in *Size[0][2*n-1]* and can be accessed in Θ(1) once the tables are filled. To know which chords are included in the final MPS, I declare another *(2n-1)*(2n-1)* table *Cases* to store each two cases: Case 1 marks when *k(the smaller endpoint) == i*, Case 2 marks when *(i < k < j) && (Size[i][k - 1] + 1 + Size[k + 1][j - 1]) >= Size[i][j - 1])*. When the specified conditions are met, Cases[i][j] is filled in with the corresponding case. This function returns a pair of objects; the first one is the max size, the second one is the `vector<vector<unsigned short>> Cases` table itself(std::pair returns by reference, so that the whole table is not copied again).

```
void Solution::RecordedChords
```

This function is used to backtrack the chords (recursion).

If Cases[i][j] is 1, it means that we encountered k == i when we were calculating the specific Max_Planar_Subset(i, j), and as a result we need to record current chord (k,j) and find those max chords within the range *(i+1, j-1)* by calling:

```
RecordedChords(i + 1, j - 1, Cases, query_data, records);
```

If Cases[i][j] is 2, by the rules we defined, we need to trace two ranges: *(k+1, j-1)* and *(i, k-1)*. I use tail recursion to be a little bit more compiler-friendly. Decrement *j* to *k-1* and the while loop will keep tracking *(i, k-1)* when the recursion of (k+1, j-1) is finished. Storing the chords with *j* and starting with higher index range ensures the chords are recorded in ascending order.

If it belongs to none of the above cases, simply decrement *j* to keep tracking down (i, j-1).

3. Flaw: Heavy Memory Usage

Because I use two tables (space complexity is $\Theta(n^2)$), and how bottom up method needs to fill in every slot, this code is very space-consuming.

When tested with self-generated test data `40000.in` (tested on eda union -p 40051), the reported memory peak is `9392056KB` (about 8G) and the runtime is 28 seconds; with self-generated `50000.in`, the reported memory peak is `58661628KB` (about 55G) and the runtime is 6 minutes. It can be concluded that this code cannot work on very big inputs otherwise it cannot have its heavy usage request satisfied. It might be better if I can come up with an algorithm that is top-down and keep space complexity in linear scale.