Algorithm HW2 PS1

Chien-I, Tseng

B11901072

1 Problem Statement

This program solves the Maximum Planar Subset problem by finding the maximum number of non-intersecting chords in a circle and listing the chords in the solution.

2 Algorithm Design

2.1 Dynamic Programming Approach

2.1.1 States Definition

1. _dp[i][j] represents the maximum number of the non-intersecting chords in the region between i and j.
2. _chord_table[i] stores the endpoint of the chord starting at point i.
3. _result stores the pairs of the enpoints representing chords in the optimal solution.

2.1.2 Base Cases

1. when i $\geq$ j, dp[i][j] = 0
2. for length 1, dp[i][i+1] = 0

2.1.3 Recursion

1. not using point j : _dp[i][j] = _dp[i][j-1]
2. if there's a chord ending at j with endpoint k = _chord_table[j]:
_dp[i][j] = max(_dp[i][j-1], _dp[i+1][j-1]) or
_dp[i][j] = max(_dp[i][j-1], _dp[i][k-1]+_dp[k+1][j-1])

2.2 Space Complexity and Time Complexity

1. Space Complexity : O(n^2)
DP table : O(n^2)
Chord table : O(n)
Result array : O(n)
2. Time Complexity : O(n^2)

Two nested loop : O(n^2)
Inside the loop : O(n^2)

3 Data Structure

3.1 Class Definition

Written in the file ./src/maxPlanarSubset.h

```cpp
class MPS {
public:
    MPS() : _num(1), _num_point(0) {}
    ~MPS() {}

    bool loadfile(const char*&);
    bool outputfile(const std::string&);
    int getnumber() { return _num; }
    int getpair(const int i, const int j);
    int mps(int, int);
    void mps_k(int, int);

private:
    int _num;
    int _num_point;
    std::vector<int> _chord_tab;
    std::vector<std::vector<int>> _dp;
    std::vector<std::pair<int,int>> _result;
    void find_solution(int i, int j);
};
```

3.2 Main DP Algorithm

```
int MPS::mps(int start, int end) {
    _dp.assign(_num_point, vector<int>(_num_point, 0));

    // Fill DP table for increasing lengths
    for (int len = 1; len < _num_point; len++) {
        for (int i = 0; i + len < _num_point; i++) {
            int j = i + len;

            // Default: don't use position j
            _dp[i][j] = (len == 1) ? 0 : _dp[i][j-1];

            // Check if we can use chord ending at j
            int k = _chord_tab[j];
            if (k >= i && k < j) {
                if (k == i) {
                    // Direct connection
                    int temp = (i + 1 < j) ? _dp[i+1][j-1] : 0;
                    _dp[i][j] = max(_dp[i][j], 1 + temp);
                } else {
                    // Connection through k
                    int temp = 1;
                    if (k > i) temp += _dp[i][k-1];
                    if (k + 1 < j) temp += _dp[k+1][j-1];
                    _dp[i][j] = max(_dp[i][j], temp);
                }
            }
        }
    }

    return _dp[start][end];
}
```

3.3 Solution Reconstruction

```
void MPS::find_solution(int i, int j) {
    if (j <= i) return;

    int k = _chord_tab[j];
    if (k < i || k > j) {
        find_solution(i, j-1);
        return;
    }

    if (k == i) {
        _result.emplace_back(i, j);
        find_solution(i+1, j-1);
        return;
    }

    // Compare using chord (k,j) vs not using it
    int score_with_k = 1;
    if (k > i) score_with_k += _dp[i][k-1];
    if (k + 1 < j) score_with_k += _dp[k+1][j-1];

    if (score_with_k > _dp[i][j-1]) {
        _result.emplace_back(k, j);
        find_solution(i, k-1);
        find_solution(k+1, j-1);
    } else {
        find_solution(i, j-1);
    }
}
```