Algorithm PA2 Report

b06502152, 許書銓

---

1. Data Structures

在這次作業裡，我使用top-down的方法寫，目的是以最省的時間完成Dynamic Programming，並且我使用到以下資料結構來儲存我所需要的資料。

a. chords

```
24      vector<int> chords;
25      int num = 0;
26      int chord_idx_1 = 0;
27      int chord_idx_2 = 0;
28      fin >> num;
29      chords.reserve(num);
30      while (fin >> chord_idx_1 >> chord_idx_2){
31          chords[chord_idx_1] = chord_idx_2;
32          chords[chord_idx_2] = chord_idx_1;
33      }
```

資料結構：vector<int>

說明：以vector存弦的端點，考量到會多次取用弦的端點，故希望取得弦端點的動作是O(n)，故以vector 來實作，輸入chords[j]會得到j的另一端點k。

b. m

```
37      int** m = new int* [num];
38      for(int i = 0; i < num; i++)
39          m[i] = new int[num];
```

資料結構：2維動態配置

說明：2維陣列，m[i][j]用來表示點i, j之間最大可以包含多少不相交弦數量，以2為動態配置實作，考量到 需要initialize花用最少的時間。

c. visited

```
41      int** visited = new int* [num];
42      for(int i = 0; i < num; i++)
43          visited[i] = new int [num];
```

資料結構：2維動態配置

說明：2維陣列，visited[i][j]用來表示，m[i][j]是否已經計算過，若是計算過則不需要再計算一次，為了實現memoization。

d. road

```
45      int** road = new int* [num];
46      for(int i = 0; i < num; i++)
47          road[i] = new int [num];
```

資料結構：2維動態配置

說明：2維陣列，road[i][j]用來表示m[i][j]是經由dp裡面的哪個case得到的，為了實現後續找弦的函式。

e. ans

```
51      vector<vector<int> > ans;
52      find_chords_topdown(0, num-1, road, chords, ans);
53      //cout << ans.size() << endl;
54      sort(ans.begin(), ans.end());
55      /*
56      for(int i = 0; i < ans.size(); i++)
57          cout << ans[i][0]  << " " << ans[i][1] << "\n";*/
58
59      /////////// write the output file //////////
60      fout << ans.size() << "\n";
61      for (int i = 0; i < ans.size(); i++)
62          fout << ans[i][0] << " " << ans[i][1] << "\n";
63      fin.close();
```

資料結構：vector<vector<int>>

說明：2維陣列，用來儲存所取用到的弦，每一個組合以vector<int>存取，再以一個vector<vector<int>>　　存取所有組合。

2. Findings

a. Top-down or Bottom-up

　在這次的PA裡，我一開始使用的是bottom-up的方式，使用完的效率大概是跑10,000筆資料時，大概需 要1左右的時間，然而在跑100,000筆資料時，所花的時間在工作站上面狀況好時(沒被砍掉時)大概要跑8 分鐘多，但如果工作站稍微塞車(也是大部分的情形)，所花的時間就會超過10分鐘。故最後我考量時間 因素，使用top-down的方式，由於top-down的算法，只會計算需要的m[i][j]，相對於計算出所有m[i] [j]，省下了大量的時間，最終實作跑10,000筆資料時大概花費0.5秒，而跑100,000筆資料時花費1分40秒 多左右，有顯著的進步。

然而，考量到找弦的方式bottom-up與top-down會有一些不同，因為在top-down裡所計算到m[i][j]的 case並不會全部都是需要加到最後要output的vector裡，故要分較多cases討論，並以back_trace的方式 找弦。然而bottom-up的情況裡因為計算出所有的cases，可以用較簡單的方式back_trace回去。

b. Initialization Matrices

在initialization所有需要的表格或是陣列時，由於不同的資料型態在initialize時間不同，舉例來說，如 果我要initialize m表格，並且所有格子裡的數值都要填0，會需要花上O(n^2)的時間，考量的這樣情形， 並且考慮使用top-down的memoization時，需要將m初始化維-1 或是 ∞，這樣會花上O(n^2)的時間複雜 度，在跑100,000筆資料時，還沒進到dp，光是initialization的時間就會爆掉了，故我採用像是BFS裡的 visited陣列，透過在創一個2維陣列，並且選擇不初始他的數字，表示visited[i][j]==0是m[i][j]還沒被計 算過，此效果等效於將m初始化到-1或是 ∞。

c. One Cpp or Mutiple Cpps

我原先在寫code時，原來將所有的functions都寫在main.cpp裡面，在同樣使用bottom-up的方式時，在10,000筆測資時，花費的時間大概是4秒；然而，經過嘗試發現將functions與main function方開來，寫 在不同cpp裡的表現會較優異，比較同樣是10,000筆資料，也是bottom-up的情況下，花費的時間大概為 2秒，有顯著時間上的差異。然而關於這方面的解釋，在網路上查詢不到太多的解釋，可能要詢問系統 相關的專業cs大師。