Algorithm Programming Assignment #2 **Report**

B07901021 潘世軒

1. **Data structure**
2. **class** ChordSet{
3. **friend** **class** Node;
4. **public**:
5. ChordSet();
6. ChordSet(**int**);
7. **bool** connection\_assign(**int**, **int**);
8. **int** find\_connection(**int**);
9. **bool** chord\_existence(**int**, **int**);
10. **void** test();
11. **private**:
12. vector<**int**> data;
14. };

使用一個叫做ChordSet的資料，將讀入的.in檔轉成可使用的集合，private member中data[i]存放和i為連線的點，舉例來說，將作業提供的12.in 讀入之後，data會變成[4, 9, 6, 10, 0, 7, 2, 5, 11, 1, 3, 8]，將資料結構這樣設計的原因是，我們可以透過端點i直接去找到他連線的另外一端點(函式find\_connection)，不需要檢查全部的點。另外，也支援判斷某兩點是否存在連線(函式chord\_existance)。

1. **Algorithm**

根據輸入建立好ChordSet後，先建立二維陣列M，其中M[i][j]用來表示M(i, j)裡的最大chord數，先將每一個數字設為-1，代表未尋訪過，然後用top-down的方式來把需要的點填滿。

接下來，由M[0][N-1]開始跑遞迴，有點像上方的走法，但是這次是用來輸出範圍內的弦，將範圍漸漸切小之後把範圍內的chord的起點慢慢寫進output檔案中。

1. **//表格的初始化和建立**
2. **int** \*\*M = **new** **int**\* [N];
3. **for**(**int** i = 0; i < N; i++){
4. M[i] = **new** **int**[N];
5. }
6. **for**(**int** i = 0; i < N; i++){
7. **for**(**int** j = 0; j < N; j++){
8. M[i][j] = -1;
9. }
10. }
12. M[0][N-1] = MPS(0, N-1, &C, M);
13. **// Topdown dynamic programming**
14. **int** MPS(**int** i, **int** j, ChordSet\* C, **int**\*\* M){
15. // cout << "fill in " << "M[" << i << "][" << j << "]" << endl;
16. **if**(M[i][j] != -1) **return** M[i][j];
17. **if**(i >= j){
18. M[i][j] = 0;
19. **return** 0;
20. }
21. **int** k = C->find\_connection(j);
23. **if**((k > j) || (k < i)){
24. M[i][j] = MPS(i, j-1, C, M);
25. }
27. **else** **if**(k == i){
28. M[i][j] = MPS(i+1, j-1, C, M) + 1;
29. }
31. **else** **if**(k < j && k > i){
32. **int** A = MPS(i, j-1, C, M);
33. **int** B = MPS(i, k-1, C, M) + 1 + MPS(k+1, j-1, C, M);
34. M[i][j] = (A >= B) ? A : B;
35. }
36. **return** M[i][j];
37. }
38. **// 跑遞迴，把在範圍內的點輸出**
39. **void** traverse(**int** i, **int** j, ChordSet\* C, fstream& fout, **int**\*\* M){
40. **if**(!M[i][j]) **return**;
41. **int** k = C->find\_connection(j);
43. **if**((k > j) || (k < i)){
44. traverse(i, j-1, C, fout, M);
45. }
47. **else** **if**(k == i){
48. fout << i << " " << j << endl;
49. traverse(i+1, j-1, C, fout, M);
50. }
52. **else** **if**(k < j && k > i){
53. **int** A = M[i][j-1];
54. **int** B = M[i][k-1] + 1 + M[k+1][j-1];
55. **if**(A >= B){
56. traverse(i, j-1, C, fout, M);
57. }
58. **else**{
59. traverse(i, k-1, C, fout, M);
60. fout << k << " " << j << endl;
61. traverse(k+1, j-1, C, fout, M);
62. }
63. }
64. }
65. **Findings**

這次作業比較困難的部分是控制記憶體用量的部分，需要填完O(n2)的表格，因此如果不好好控管記憶體的話很容易就會炸開，再來在填表格的時候後來選用Top down的方法，因為這次作業要求的只有找出0~N-1範圍內的點，不需要多次計算，因此選用top down的方法可以省時間，最後100000筆的測資可以壓在大約兩分鐘之內跑完，算是還蠻OK的。   
另外vector雖然很好用，但是他在自己分配空間(一個一個pushback)的時候是直接無腦兩倍的樣子，因此用比較古老的方法(用指標陣列，自己new空間出來)可能會對記憶體用量有比較多的掌控度，另外根據有修計算機結構的同學說法，這樣似乎也會比較快