Algorithms PA2 Report

b07901033

1. Program Flow

The main program can be divided into 4 parts. First, read input file and parse it. Second, filling up the table by dynamic programming to find the maximum planar set. Third, backtrack the chords contained in my solution. Last, write the result in the output file.

The following is the second part.

Algorithm 1 Find_MPS(C,N)

for $d = 1$ to $2N$-1 **do**
  for $i = 0$ to $2N$-d-1 **do**
    $j = i + d$
    if $chord\ kj \in C\ and\ k == i$ **then**
      $M(i,j) = M(i+1, j-1) + 1$
    else if $chord\ kj \in C\ and\ k \notin [i,j]$ **then**
      $M(i,j) = M(i, j-1)$
    else
      $M(i,j) = \max[M(i,j-1), M(i,k-1) + 1 + M(k+1, j-1)]$
    end if
  end for
end for

The following is the third part.

Algorithm 2 Backtrack_MPS(C,N)

declare a stack : tmp_stack
declare an array : result[M(0,2*N-1)]
tmp_stack.push(0,2N-1)
while Not($tmp\_stack.enpty()$) **do**
  $p = tmp\_stack.pop()$
  $i, j = p[0], p[1]$;
  if $chord\ kj \in C\ and\ k == i$ **then**
    $result.pushBack(i)$
    $tmp\_stack.push(i+1, j-1)$
  else if $chord\ kj \in C\ and\ k \notin [i,j]$ **then**
    $tmp\_stack.push(i, j-1)$
  else
    if $M(i,j) == M(i, j-1)$ **then**
      $tmp\_stack.push(i, j-1)$
    else
      $result.pushBack(k)$
      $tmp\_stack.push(i, k-1)$
      $tmp\_stack.push(k+1, j-1)$
    end if

The reason for using stack and the detail implementation will be discussed later.

2. Time optimization

In order to speed up the execution time, I use classic c++ arrays without dynamically allocating memory.

The structure of "Find_MPS(C,N)" needs at least N^2. So I must speed up the time for searching chord kj. I create an array "chords_k" of size $2N$ to store the information of chords. "chords_k[i]=j" means that there exists a chord ij. By this delicate design, I can view "chords_k" as a function to find the other correspondent endpoint of the given index. In addition, this process is constant time, since it is a random access array. Hence, by careful analysis, the job done in each iteration is constant time, which makes my program more efficient. So does "Backtrack_MPS".

3. Space optimization

Due to the large data input, I give up using 2d array of size $2N \times 2N$ for my dp table. 2d array is convenient for writing code. However, in this program, we only use up approximately half of the 2d array, which is a considerable waste as N grows. Instead of 2d array, I use an 1d array of total size $N(2N-1)+1$. Also, I define a function for converting (i,j) pair to actual array index."$index = j(j-1)/2 + i + 1$" The "1" was added at the end of the equation for reserving the first item to be 0. i.e array[0]=0. This is designed for writing convenience. If i equals j, the conversion function can simply just return 0, fitting perfectly in "dp[conversion_function(i,j)]=0"

Another Space optimization is backtracking. I choose not to record the steps derived during "Find_MPS(C,N)", due to huge memory overhead. Instead, I actually find the chords by top-down method, starting from (0,2N-1). To prevent recursive function calling, I use stack to do the job. I get this idea from BST traversal.

4. Results

Table 1. Execution results

x	y
Elapsed Time	1m29.135s
User Time	1m25.516s
System Time	0m3.628s
Space	19G