Project 2

Safe Fruit

Meng JunYi

Date: April 11, 2021

Chapter 1 : Introduction

## 1 Problem Description

Some fruits must not be eaten with some other fruits, otherwise it is easy for us to get into trouble. Now given a long list of tips each of which tells us a pair of fruits that cannot be eaten at the same time and a big basket of fruits, our goal is to find out the maximum subset of fruits that is safe to eat any of them at the same time.

## 2 Input Specification

First line gives us two positive integers: N, the number of tips, and M, the number of fruits in the basket. both numbers are no more than 100. Then there followed N pair of fruits which must not be eaten together, and M fruits together with their prices. Clearly, each fruit is represented by a 3-digit ID number. A price is a positive integer which is no more than 1000.

## 3 Output Specification

The output ought to have three lines. First line gives the maximum number of safe fruits. Second line list all the safe fruits, in increasing order of their IDs. And the last line prints the total price of the above fruits. When two or more collection of fruits share the same volume, output the one with the lowest total price.

Chapter 2: Algorithm

Data Structure: Graph

We use the adjacency matrix to represent the graph G = (E, V), where the vertex V represents fruit, and the edge E connects fruits that cannot be eaten at the same time.

Initially, the adjacency matrix is (All zeros):

The following pseudo code illustrates the process of reading the graph.

Algorithm: Native DFS + Backtracking

Input: N - number of tips; G - two dimensional array

function ReadGraph(N, G)

for cnt ← 0 to N

scanf i, j;

Graph[i][j] = Graph[j][i] = 1;

end for

end function

We use depth-first-search(DFS) to find the maximum independent set.

The following pseudo code illustrates how native DFS is carried out.

Note: the definition of cur in the pseudo code below is like a iterator, when it reaches the end, it ought to be None.

Global: G - two dimensional array, S – collection of selected fruit

Max – maximum number of fruits, Min – min cost for the fruits

Input: cur – current fruit; cost – total price of currently selected fruits

num – number of currently selected fruits

function dfs(cur, cost, num)

*# Reach a leaf node*

if cur == None then

if S.isLegal() then

if Max < Num then

Max = num

Min = cost

end if

if Max == Num and cost < Min then

Min = cost

end if

end if

return

end if

S.append(cur)

dfs(cur.next(), cost + cur.price(), num + 1)

S.remove(cur)

*# Backtracking*

dfs(cur.next(), cost, num)

end function

Comment:

S.isLegal() checks if there exists a pair of fruits in S that must not be eaten together.

This version of DFS does not carry out any pruning, it just goes deeper, deeper until it satisfies the terminate condition (can’t go any deeper), so it will visit all the leaf nodes (including those have contractionary fruits), absolutely.

Algorithm: DFS + Backtracking + Pruning

Simply using DFS cannot pass the test. We need to pruning the search tree to reduce the running time.

Global: G - two dimensional array, S – collection of selected fruit

Max – maximum number of fruits, Min – min cost for the fruits

Input: cur – current fruit, cost – total price of currently selected fruits

num – number of currently selected fruits

function dfs(cur, cost, num)

i = cur

while i

if num + M - i < Max ① or num + res[i] < Max ②

return

for fruit in S

if G[i.ID()][fruit.ID()] goto next

end for

S.append(i)

dfs(i.next(), cost + i.price(), num + 1)

S.remove(i)

next:

i = i.next()

end while

if Max < Num then

Max = num

Min = cost

end if

if Max == Num and cost < Min then

Min = cost

end if

end function

Global: M – M kinds of fruits, res – dict that used to store MAX from i ,

Max – maximum number of fruits, Min – min cost for the fruits function maxIndependSet()

for i ← 0 to M

Max = 0

Min = INFINITY

dfs(i, 0, 0);

res[i] = MAX;

end for

Comment:

res is maintained for later called dfs to pruning.

end function

Explanation:

There are two cases that pruning shall occur:

1. First occasion is easy to think. We can predict the final number of fruits selected each time entering a DFS. In ideal case, the rest of the fruits have no conflict between each pair and have no conflict with all the fruits we have currently selected. So when the size of the currently selected subset adding all the fruits remaining is still smaller than the optimal solution that has been found, then there is no inevitable search and pruning can be done. I have labeled it ① in the pseudo code above.
2. However only using the first skill to pruning is not enough, that is because the condition is too strong, in most cases, the left unsearched part of fruits have pairs that can’t be eaten together. Therefore, if we make the condition weaker, more pruning will occur thus reduce the time complexity. We can record the maximum number of fruits that can be eaten together from fruit to fruit. Next time when entering a search node that is at fruit, we simply compare the currently selected numbers adding that record with currently maximum number, if less than, then there is no possibility better selection can be found in all its subtrees, that is where I labeled ② in the pseudo code.

However, there still exists one problem, how can we know the maximum number of fruits that can be eaten together from fruit to fruit? In implementation, I use an idea that is similar to dynamic programming. I define an array ***res*** to record the size of largest group from fruit to fruit. ***res[i]*** stores the maximum number of fruits in [fruits[i] fruits[i+1] … fruits[m] ]. Since we can specify the initial search position of DFS algorithm, ***res[m]*** can be achieved by calling dfs(m, 0, 0). And then ***res[m]*** is used to accelerate dfs(m-1, 0, 0), then ***res[m], res[m-1]*** is used to accelerate dfs(m- 2, 0 ,0)… Finally when dfs(0, 0, 0) is called, final answer is obtained.

Chapter 3: Testing Results

Case 1: Sample Input

|  |  |
| --- | --- |
| Input | Output |
| 16 20  001 002 003 004 004 005 005 006 006 007 007 008 008 003 009 010 009 011 009 012 009 013 010 014 011 015 012 016 012 017 013 018  020 99 019 99 018 4 017 2 016 3 015 6 014 5 013 1  012 1 011 1 010 1 009 10 008 1 007 2 006 5 005 3 004 4 003 6 002 1 001 2 | 12  002 004 006 008 009 014 015 016 017 018 019 020  239 |

For the sample case, it must be right.

Case 2: Sample Input with unordered price list

|  |  |
| --- | --- |
| Input | Output |
| 16 20  001 002 003 004 004 005 005 006 006 007 007 008 008 003 009 010 009 011 009 012 009 013 010 014 011 015 012 016 012 017 013 018  005 3 019 99 009 10 017 2 011 1 015 6 014 5 013 1  012 1 016 3 010 1 018 4 008 1 007 2 006 5 020 99 004 4 003 6 001 2 002 1 | 12  002 004 006 008 009 014 015 016 017 018 019 020  239 |

Still output the selected fruits in increasing order, this is right.

Case 3: No fruit can be eaten together

|  |  |
| --- | --- |
| Input | Output |
| 3 3  001 002  002 003  003 001  002 98  001 99  003 3 | 1  003  3 |

As expected, it print out the only fruit with lowest price.

Case 4: All fruits can be eaten together

|  |  |
| --- | --- |
| Input | Output |
| 0 11  001 1 002 2 003 3 004 4 005 5 011 999 010 998 009 997 008 996 007 995 006 994 | 11  001 002 003 004 005 006 007 008 009 010 011  5994 |

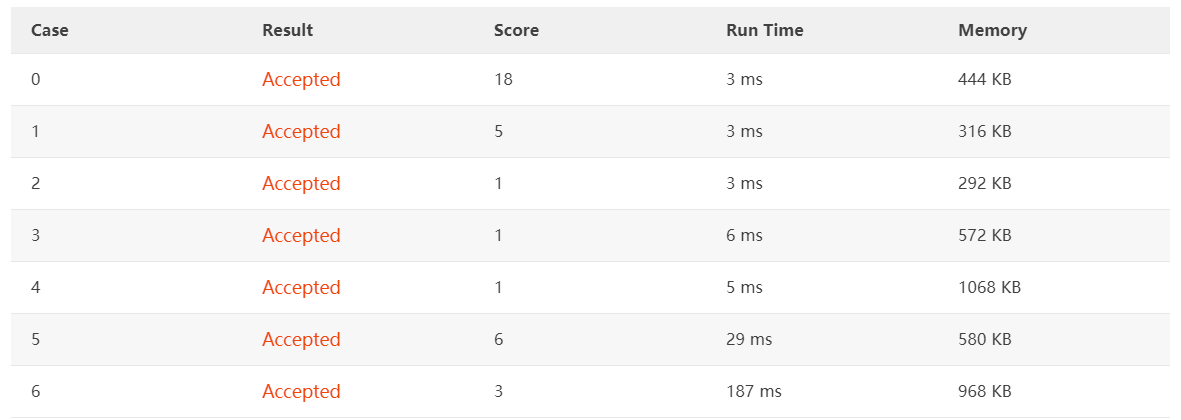
In this case, no tip is provided, all the fruits can be eaten at the same time, like what we expect, the program output all the fruits in increasing order.

Case 5: Maximum data

|  |  |
| --- | --- |
| Input | Output |
| See in ../testcase/case5.txt | 16  002 033 060 089 425 472 546 557 566 587 604 670 687 713 816 894  8219 |

In the testcase folder, I wrote a generator to produce random test data, input and , it can produce data of that scale. It takes a bit of time to calculate this case, but finally gives us the right answer.

PTA Testing Result



Chapter 4: Analysis and Comments

Analysis

It is known that the maximum independent set problem of graphs is equivalent to the maximum clique problem. However, both are proved to be NP-Complete questions, so it is impossible to solve them in polynomial complexity algorithm. Only searching can gives us the right answer, and pruning is a powerful tool to speed up it.

Suppose there are tips, kinds of fruits, the maximum ID for a fruit is .

Time complexity

Firstly, we ought to read tips and store them in Graph, time complexity for this part is obviously .

Secondly, since the best answer is needed, a simple native DFS algorithm should exhaust all the possibilities. Imaging we are using bit string to encode our selection, for example (1 0 0 0 …) means we only collect first fruit into the set, knowing that are kinds of fruit, it is clearly that there exists such distinct selection. So, there are lead nodes in the search tree for this question. So DFS is supposed to be called ( + -1) times. Any way, it is still ). What is more, don’t forget we ought to traverse the tip list to see if a conflict occurs when collecting the current fruit into the set in every DFS call unless we are visiting a leaf node, that requires time. So the total time complexity is

**Taking the two parts above into account, the global time complexity is ).**

According to the analysis above, this seems quite unacceptable for large scale of data, however with the help of pruning, we can pass all the testing point.

Space complexity

In the complementation, we use adjacency matrix to store the graph, so our space complex is

Comments:

1. In my opinion, the general idea is quite similar to dynamic programing. But we don’t directly use the answer of sub-question to calculate the final answer, instead they are used for pruning thus accelerate the whole process.
2. One famous algorithm for this type of question is Bron-Kerbosch algorithm. Anyway, the basic ideas are pretty similar, both dfs and Bron-Kerbosch require backtracking. In fact, the Bron-Kerbosch algorithm is originally an algorithm which is only a recursive backtracking algorithm. In simple cases, Bron-Kerbosch algorithm performs slower, but for dense graph, optimized Bron-Kerbosch algorithm is the better choice.

Declaration

**I hereby declare that all the work done in this project titled "Safe Fruit" is of my independent effort.**

Appendix: Source Code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 1000

#define INF 1e9

int Price[MAX];

int G[MAX][MAX];

// IndexToID

int ID[MAX];

// temp data need for dfs

int selc[MAX];

// res records the maximum fruits from subset which index from i to m

int res[MAX];

// final answer to the problem

int max\_sel;

int min\_price;

int final\_selc[MAX];

// n tips, m kinds of fruit

int n, m;

// Used for comparsion in quick sort

int cmp(const void \*a, const void \*b){

    return \*(int\*)a - \*(int\*)b;

}

// When the start location is different

// Everytime maxIndependSet calls dfs, clean previous data first

void Init(){

    max\_sel = 0;

    min\_price = INF;

}

// Save the current best choices

void copy\_selc(){

    memset(final\_selc, 0, MAX \* sizeof(int));

    int count = 0;

    for (int i = 0; i < m; i++){

        if(selc[i]){

            final\_selc[count++] = ID[i];

        }

    }

}

// DFS, calc maximum number of fruits in set [cur, cur+1, cur+2, ..., m-1]

void dfs(int cur, int price, int count){

    for(int i = cur; i < m; i++){

        /\* four cases that pruning should occur:

        1. Adding all the fruit that can be eaten together is still smaller th an the current best choice

        2. Adding all the fruit left is still smaller than the current best choice

        3. Adding all the fruit left can't produce better cost

        4. Adding all the fruit that can be eaten together can't produce better cost

        \*/

        if  (count + res[i] < max\_sel ||count + m - i < max\_sel

            ||(count + m - i == max\_sel && price >= min\_price)

            ||(count + res[i] == max\_sel && price >= min\_price)) return;

        // Check if there exists a conflict with previous selected fruits, if so, move to next fruit

        for (int j = 0; j < m; j++){

            if (selc[j]){

                if(G[ID[j]][ID[i]]) goto next;

            }

        }

        selc[i] = 1;

        // Pass the new price and let number of fruits selected plus one

        dfs(i + 1, price + Price[ID[i]], count + 1);

        // Backtracking

        selc[i] = 0;

        next:;

    }

    // Check and update the optimal solution

    if(count > max\_sel){

        max\_sel = count;

        min\_price = price;

        copy\_selc();

    }else if(count == max\_sel && price < min\_price){

        min\_price = price;

        copy\_selc();

    }

}

void maxIndependSet(){

    for(int i = m - 1; i >= 0; i--){

        Init();

        dfs(i, 0, 0);

        // Save the maxIndependSet

        res[i] = max\_sel;

    }

}

int main(int argc, char const \*argv[]){

    int i, j;

    scanf("%d %d", &n, &m);

    for (int c = 0; c < n; c++){

        scanf("%d %d", &i, &j);

        G[i][j] = 1;

        G[j][i] = 1;

    }

    for (int c = 0; c < m; c++){

        scanf("%d %d", &i, &j);

        ID[c] = i;

        Price[i] = j;

}

    // Make ID increasing to simplify final print step

    qsort(ID, m, sizeof(int), cmp);

    maxIndependSet();

    printf("%d\n",max\_sel);

    for (i = 0; i < max\_sel - 1; i++){

        printf("%03d ", final\_selc[i]);

    }

    printf("%03d\n", final\_selc[max\_sel-1]);

    printf("%d\n",min\_price);

    return 0;

}