Safe Fruit

Group 18

Date:2020-04-12

CONTENT

Chapter 1 Introduction

1.1 Background

There are some kinds of fruits, but some of them can't be eaten together, if so, there could be some serious trouble. Now we are given the fruits and required to select the most kinds that can be eaten together safely with the least cost.

1.2 Problem Description

1.2.1 Input Specification

Each input file contains one test case. For each case, the first line gives two positive integers: N, the number of tips, and M, the number of fruits in the basket, both numbers are no more than 100, like the following format:

```
FruitID1 FruitID2
```

Then two blocks follow. The first block contains N pairs of fruits which must not be eaten together, each pair occupies a line and there is no duplicated tips; and the second one contains M fruits together with their prices, again each pair in a line. To make it simple, each fruit is represented by a 3-digit ID number. A price is a positive integer which is no more than 1000. All the numbers in a line are separated by spaces, like following format:

```
FruitID Price
```

1.2.2 Output Specification

In the first output line, we are asked to provide the number of fruits that can be eaten together safely.

In the second output line, we are asked to provide the answer with the most variety of fruits included that can be eaten together safely among the list of fruits given before.

And in the last output line, we should give the total price of the fruits listed in the second line.

1.2.3 Problem Analysis

The problem can be abstractedly described as a *Maximum Clique* problem, which is going to find the maximum complete subgraph of a graph(or to find the maximum independent set of the complement graph of this graph, which is the way we choose). In this problem, all fruits constitute one graph and each kind of fruit is a vertex. If two fruits cannot be eaten together, then there is an edge between them, otherwise there is no edge between these two fruits. In this way, the problem can be simplified into the problem to find the maximum independent set of this graph.

Chapter 2 Algorithm Specification

2.1 Backtracking Algorithm Combined With Pruning

Backtracking means the method of exhaustively multi-dimensional data can be thought of as multi-dimensional Exhaustive Search.

The general idea is: treat multi-dimensional data as a multi-dimensional vector (solution vector), and then use recursion to sequentially exhaust the values of each dimension to produce all possible data (solution space), and on the way back Avoid listing incorrect data.

The pseudocode of backtracking is:

```
int solution[MAX_DIMENSION];     /* solution vector*/

void backtrack(int dimension){
    /* Produced a set of data, and verified that this set of data is incorrect*/
    if (solution[] is well-generated){
        check and record solution;
        return;
    }

    /* Exhaustively enumerate all the values of this dimension and return to the next dimension*/
    for (x=each value of current dimension){
        solution[dimension]=x;
        backtrack(dimension+1);
    }
}
```

The backtracking method will avoid enumerating incorrect data on the way back, and its meaning is actually equivalent to the pruning technology of the search tree.

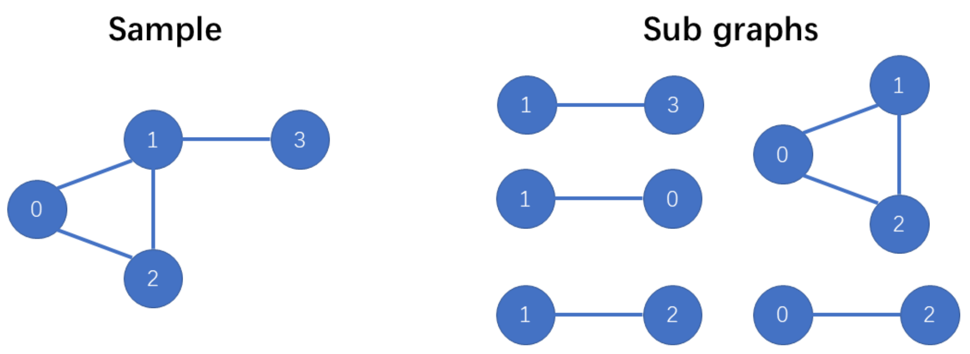
```
int solution[MAX_DIMENSION];
void backtrack(int dimension){
    /* pruning: Avoid listing incorrect data on the way back*/
    if(solution[] will NOT be a solution in the future)  return ;
    if(solution[] is well-generated){
        check and record solution;
        return;
    }

    for(x=each value of current dimension){
        solution[dimension]=x;
        backtrack(dimension+1);
    }
}
```

2.2 Bron-Kerbosch Algorithm(Maximum Clique)

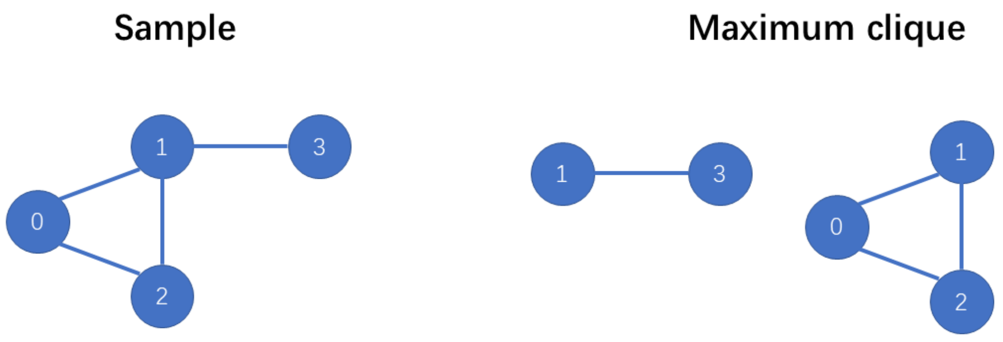
The simple idea to find the Maximum Clique is:

1. Generate all subgraphs of the original graph (there may be 2n-1 subgraphs, n represents the number of nodes);
2. Determine whether these subgraphs are a group;
3. Delete all groups that are not great groups;

For example, there are many subgraphs in the following sample graph.



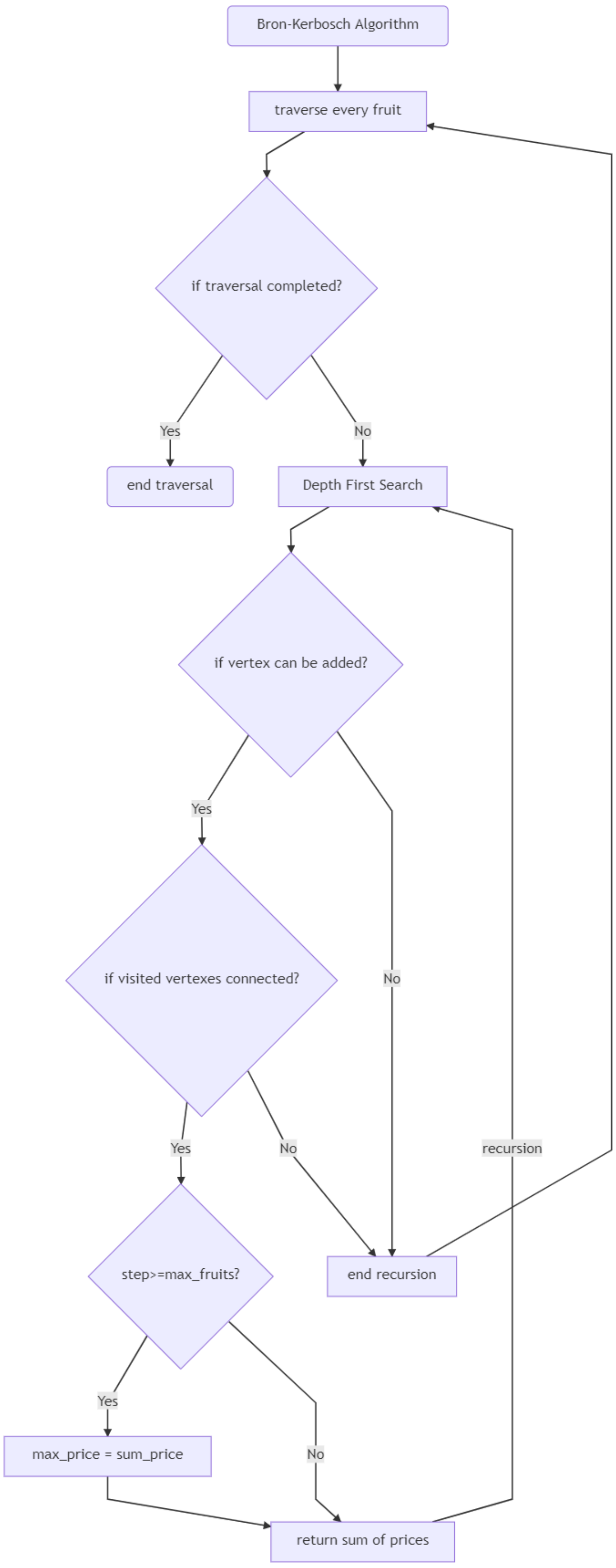
In these subgraphs, we will find that some are not cliques, for example, the graph composed of {1, 2, 3}, because 2 and 3 are not connected, so such a graph does not meet the requirements, remove it. All the remaining subgraphs are those that meet the requirements of the regiment. Finally, there are many of them, which are not maximal cliques. For example, {0, 1} is not a maximal clique because of the existence of {0, 1, 2}. These are not groups of great groups, and we have to remove them. Now the rest is what we are looking for.



The pseudocode of the Bron-Kerbosch Algorithm is:

```
Bron-Kerbosch Algorithm(Version 1)
R={}   /*The set of vertices of the determined maximal clique*/
P={v}  /*Unprocessed vertex set, initial state is all nodes*/
X={}   /*A set of vertices that have been searched and belong to a certain maximal group*/

BronKerbosch(R, P, X):
   if P and X are both empty:
       report R as a maximal clique
   for each vertex v in P:
       BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
       P := P \ {v}
       X := X ∪ {v}
```

The sketch of the algorithm is:


Bron-Kerbosch Algorithm
traverse every fruit
if traversal completed?
Yes
No
end traversal
Depth First Search
if vertex can be added?
Yes
No
if visited vertexes connected?
Yes
No
recursion
step>=max_fruits?
end recursion
Yes
No
max_price = sum_price
return sum of prices

Chapter 3 Testing Results

3.1 Testing Results in Total

Index	Test file name	Testing intention	Result	Run Time
1	test1.txt	sample input	Correct	0.007s
2	test2.txt	minimal size of input data, without loop	Correct	0.002s
3	test3.txt	minimal size of input data, with loops	Correct	0.003s
4	test4.txt	medium size of input data, sparse graph	Correct	93.658s
5	test5.txt	medium size of input data, dense graph	Correct	0.004s
6	test6.txt	medium size of input data, complete graph	Correct	0.003s
7	test7.txt	medium size random input data	Correct	0.183s
8	test8.txt	medium size random input data	Correct	0.022s
9	test9.txt	maximal size of input data	Timeout	--

3.2 Testing Results in Details

Test 1 sample input

```
//output
12
002 004 006 008 009 014 015 016 017 018 019 020
239
```

This test uses the sample input, and the output is the same with the sample output.

Test 2 minimal size of input data, without loop

```
//output
1
002
3
```

This test consists of one tip and two kinds of fruits, which is the minimal input with no loops. The output is correct.

Test 3 minimal size of input data, with loops

```
//output
1
002
4
```

This test consists of three tips and three kinds of fruits, which is the minimal input with a loop. The output is correct.

Test 4 medium size of input data, sparse graph

```
//output
63
001 002 003 004 005 007 008 009 010 011 012 014 015 016 017 018 019 020 021 022 023 024 025 026 027
028 029 031 033 034 035 036 038 039 040 041 042 043 045 047 048 049 050 052 053 056 057 058 059 061
063 064 065 066 067 068 069 070 071 072 073 074 076
24474
```

This test consists of 20 tips and 76 kinds of fruits, which means it is a sparse graph whose density is $20 \div 2850 = 0.701\%$. Though costs a little long time, is still gives the correct answer.

Test 5 medium size of input data, dense graph

```
//output
4
004 007 010 011
2124
```

This test consists of 50 tips and 14 kinds of fruits, which means it is a dense graph whose density is $50 \div 91 = 54.945\%$. The output is correct.

Test 6 medium size of input data, complete graph

```
//output
1
009
48
```

This test consists of 91 tips and 14 kinds of fruits, which means it is a complete graph (whose density is 100%). The output is correct.

Test 7 medium size random input data

```
//output
33
001 002 003 004 008 009 012 013 015 016 017 018 019 021 022 023 025 029 030 031 033 035 036 037 038
039 040 043 046 047 048 049 050
16382
```

This test consists of 45 tips and 50 kinds of fruits, and the output is correct.

Test 8 medium size random input data

```
//output
17
003 005 011 012 013 014 016 017 019 020 021 025 027 030 031 032 037
8243
```

This test consists of 84 tips and 38 kinds of fruits, and the output is correct.

Test 9 maximal size of input data

This test consists of 100 tips and 100 kinds of fruits, whose density is $100 \div 4950 = 2.02\%$. Since the input graph is an extremely large sparse graph, the program takes about 8 hours while still has 19 loops to run. Surely the program will give the correct answer in the end, while the running time is long enough.

Chapter 4 Analysis and Comments

4.1 Analysis on Time Complexity

We're trying to use Bron-Kerbosch Algorithm to solve this Maximal Clique Problem. Since maximum clique calculating costs a lot of time on sparse graphs, we traverse indexes decreasingly to improve the Bron-Kerbosch Algorithm. However, the algorithm is not an output-sensitive algorithm and it does not run in polynomial time. The running time is related to the specific input. For small

Through further searching and learning, we get to know that Maximal Clique Problem is a Non-deterministic Polynomial Complete problem. Those sparse graphs with many vertexes have the exponential growth of running time.

4.2 Analysis on Space Complexity

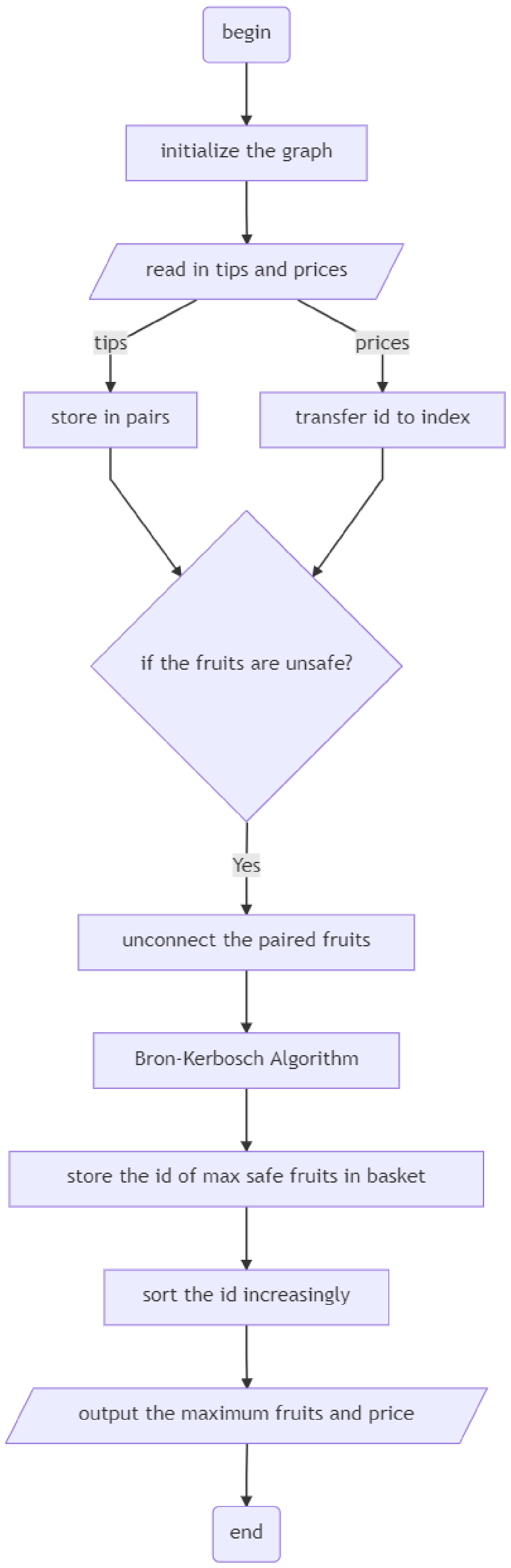
Using Adjacent Matrix to save the data of graph, the space complexity is $O(N^2)$, where N is the number of the maximal kinds of fruits.

4.3 Comments

In conclusion, the Bron-Kerbosch Algorithm works well in small input. When handling extreme conditions like sparse large graph, it will take a long time because the recursive depth increases rapidly even if we use prune method.

Appendix

Sketch of the Program



Source Code in C++

```cpp
#include <bits/stdc++.h>
using namespace std;
#define MAXVERTEX 1005
#define MAXNUM 105
int num_fruits, num_tips, max_fruits = 0, max_price = 0;
//the number of tips and fruits in the basket, that is, N and M
int graph[MAXNUM][MAXNUM];
int price[MAXNUM], visit[MAXNUM], safe_fruits[MAXNUM];
//store the price of each fruit
int dp[MAXNUM];
//dp[i] is the number of nodes in the maximum clique from i to num_fruits
int id2index[MAXVERTEX], index2id[MAXNUM];
//id2index[id]=index, index2id[index]=id
vector<int> max_basket;
//store the index of max safe fruits in the bastet
vector<pair<int, int>> tips(MAXNUM);
//store the unsafe fruits in pair
void dfs(int i, int sum_price, int step);

int main()
{
    int i, j;
    //initialize the graph (build graph)
    memset(graph, 1, sizeof(graph));

    //start input
    //freopen("test.txt", "r", stdin);
    cin >> num_tips >> num_fruits;
    //read in tips
    for (i = 0; i < num_tips; i++)
        cin >> tips[i].first >> tips[i].second;
    //read in prices
    for (i = 0; i < num_fruits; i++)
    {
        int id, temp_price;
        cin >> id >> temp_price;
        //note that the index and id both start from 1
        id2index[id] = i + 1;
        index2id[i + 1] = id;
        price[id2index[id]] = temp_price;
    }
    //unconnect the unsafe fruits
    //note that the graph's vertex is represented by index
    for (i = 0; i < num_tips; i++)
        graph[id2index[tips[i].first]][id2index[tips[i].second]] =
graph[id2index[tips[i].second]][id2index[tips[i].first]] = 0;

    //Bron-Kerbosch Algorithm(Maximum Clique)
    for (i = num_fruits; i > 0; i--)
    {
        visit[0] = i;
        //traversal
        dfs(i, price[i], 1);
        dp[i] = max_fruits;
    }

    //store the id of max safe fruits in max_basket
    for (i = 0; i < max_fruits; i++)
        max_basket.push_back(index2id[safe_fruits[i]]);

    //sort the index of fruits increasingly
    sort(max_basket.begin(), max_basket.end());

    //output
    cout << max_fruits << endl;
    for (i = 0; i < max_fruits - 1; i++)
        printf("%03d ", max_basket[i]);
    printf("%03d\n", max_basket[max_fruits - 1]);
    cout << max_price << endl;

    system("pause");
    return 0;
}

void dfs(int i, int sum_price, int step)
{
    int m, n;
    //enumerate each node and record those connected with larger index
    for (m = i + 1; m <= num_fruits; m++)
    {
        int flag = 0;
        if (dp[m] + step < max_fruits || (dp[m] + step == max_fruits && sum_price >= max_price))
            //pruning
            return;      //vertex cannot be added to the clique anymore
        if (graph[i][m]) //i and m are connected
        {
            for (n = 0; n < step; n++)
                if (graph[m][visit[n]] == 0)
                {   //jump over the loop if m and n are not connected
                    flag = 1;
                    break;
                }
            if (!flag)
            {
                visit[step] = m;
                dfs(m, sum_price + price[m], step + 1);
                //process the nodes recursively
            }
        }
    }
    //return the maximum results if step>=max_fruits
    if (step > max_fruits || (step == max_fruits && sum_price < max_price))
    {
        for (m = 0; m < step; m++)
            safe_fruits[m] = visit[m];
        max_fruits = step;
        max_price = sum_price;
    }
}

```

Testing Set Generating Script in PHP

```php
<?php
$graph = array();
$M = mt_rand(1,100);
$Mmax = $M*($M-1)/2;
$N = mt_rand(1,100);
while ($N > $Mmax) $N = mt_rand(1,100);
echo $N." ".$M."\n";
for ($i = 0; $i < $N; $i++) {
    $id_1 = mt_rand(1,$M);
    $id_2 = mt_rand(1,$M);
    if ($id_1 == $id_2){
        $i--;
        continue;
    }
    $check1 = str_pad($id_1,3,"0",STR_PAD_LEFT)."->".str_pad($id_2,3,"0",STR_PAD_LEFT);
    $check2 = str_pad($id_2,3,"0",STR_PAD_LEFT)."->".str_pad($id_1,3,"0",STR_PAD_LEFT);
    if(in_array($check,$graph)||in_array($ancheck,$graph)){
        $i--;
        continue;
    }
    $graph[] = $check1;
    $graph[] = $check2;
    printf("%03d %03d\n",$id_1, $id_2);
}
for ($i = $M;$i > 0; $i--) {
    $price = mt_rand(1,1000);
    printf("%03d %d\n",$i,$price);
}
?>
```

Declaration

We hereby declare that all the work done in this project titled "Safe Fruit" is of our independent effort as a group.

Duty Assignments

Programmer:
Tester:
Report Writer: