Project 2 - Safe Fruit - Report

Group 5 (Zhai, Duan and Chen) Date: 2021-4-16

Introduction

Background

Some fruits cannot be eaten together, in case causing health problems. Now we're given the tips that which pair of fruits cannot be eaten together, and a number of fruits, each of which is of a 3-digit ID and a positive price. And we want to find out the maximum number of fruits that can be eaten together. If there's a tie, we want to find out the one with the lowest price.

Input Specification

The first line consists of two positive interger $N, M (N, M \leq 1000)$, split by a space. N is the number of tips and M is the number of fruits **available。** Each fruit is identified by a 3-digit ID.

The following N lines are the tips, each of which is a pair of ID split by a space, representing two fruits that must not be eaten together. There's no duplicated tips.

This is followed by M lines. Each line consists of a fruit ID, then a space, and then a positive number p (≤ 1000), the price of the fruit.

Output Specification

First print in a line the maximum number of fruits that can be eaten together. In the next line list all the fruit ID in ascending order of their IDs. The ID's must be separated by exactly one space, with no extra space at end of the line. The third line should consists of the total price of the above fruits.

Hints

1. **The solution is guaranteed to be unique**
2. **According to the description**, only the fruits with a price known can be chosen; not all the fruits given by the whole input can be chosen。

Algorithm Specification

Overview

The algorithm consists of the following three steps:

1. Read in the data.
2. Search the solution space by backtracking to find out the unique solution ;
3. Print out the solution.

The second step is accomplished by the *dfs* function by **recursion** and is optimized through **pruning**.

Data Structure

The following struct is used in the code:

```
struct fruit_array{
    int p, value;
    int fruit[MAXN];
} bestBasket, curBasket, fruits[MAXID];
```

The struct works in different ways under different circumstances, where the semantic of its members are differs.

1. It can store the information of a specific fruit, including its price `value`, the number of fruits that cannot be eaten with it (we will call it **unsafe** then) `p` and all the indices of them in an array `fruit[]`. The information of all these fruits is stored in the array `fruits[]`.
2. It can also store the information of the **basket** (a collection of fruits). In this case, `p` represents including the number of fruits in the basket, `fruit[]` stores their indices and `value` stands for their total price. As instances, respectively, `bestBasket` and `curBakset` represents the currently best solution and the current basket in the searching procession.

Some other important macro definitions and their meanings are as follows:

name	description
MAXN	The upper bound of the number of the fruits
MAXID	The upper bound the ID of the fruits
mark[i]	标记`curBasket`中与编号为i的水果排斥的水果种数
dp[i]	从编号$i \sim N$的水果中满足条件的组合中水果种数的最大值
ind[i]	编号为i的水果的ID

Each fruit is identified in two ways in the code, one the **3-digit** ID, the other a **index** ranging from 1 to N, which is allocated according to the ID, in ascending order (which means the fruit with the smallest ID should be allocated the index 1, and the largest ID should correspond to index N).

Most of the arrays in the program uses ID as their indices, while the argument of *dfs* takes an index. Therefore, an array `ind`, converting the ID to its corresponding index, is needed.

Data Read-in

This stage comprises the following steps:

- **Read in all the tips.** For each pair of fruits, only the fruit with the larger index is pushed into the list the other fruit, since a sorting process afterwards will guarantee that while searching, if the larger-index fruit is being considered, the smaller-index fruit has already been considered.
- **Read in the prices**. Besides this, we should also record all the indices of the fruits available into the `ind` array that will be sorted in the next step.
- **Sorting**. Sort `ind` in ascending order. Given that the number of fruits is rather small (not over 100) and hence the sorting is not the mostly time-consuming process, a simple **selection sort** is applied, and implemented in the *min_sort* function.

The *dfs* Function

dfs function deploys a depth-first-search strategy and is optimizied by pruning through several ways.

dfs* calls in *main

```
for(int i = N; i > 0; i--){
    dfs(i);
    dp[i] = maxPath.p;
}
```

Here we call `dfs[i]` reversely in a `for` structure. The i-th iteration finds out the maximum number of safe fruits with indices ranging from i to N, which is stored in `dp[i]` and used for pruning.

the body of *dfs*

`dfs[i]` does the following things for each of the fruits possible to be picked up into the basket（i.e., with a$i \sim N$ index）：

- Decide if reducing to the boundary conditions. If so, start backtracking, after deciding whether the `curBasket` is a better solution than `bestBasket` and update the latter if true.
- Decide if a pruning is allowed. If so, start backtracking.
  The method is to first calculate the diffence of the number of fruits in `bestBasket` and `curBasket`, and then compare it with the `dp[i+1]`, the possible maximum number of safe fruits that can be picked up among the fruits left. If the latter, plus one, is smaller than the former, then no matter how we choose the fruits out of the fruits left, it's impossible for us to reach a better solution with a larger number of safe fruits. So we can end our searching here. This corresponds to the following line of code:

  ```
  if (dp[i+1] + 1 < maxPath.p - curPath.p) break;
  ```

- Decide whether the fruit is available and whether it's still safe if the fruit is added into the basket (by deciding if `value` and `mark` are above zero respectively). If the fruit is unavailable or unsafe, we simply skip it.
- Add the fruit into the basket, update the number `p` and total price `value` of the basket and add one to the `mark` of all the fruits that cannot be eaten with it. Then we call `dfs[j+1]`, where `j` is the index of the current fruit being considered.
- When backtracking from the lower level, undo all the operations in the previous step that are done before calling `dfs[j+1]` .

Testing Results

Sample Input

No.	Description
1	16+20，测试样例，编号恰为1~20
2	1+2，最小样例，大编号水果更便宜
3	1+2，最小样例，小编号水果更便宜
4	4+8，存在水果种数相同的解
5	9+6，二分图，两部分点数量相同
6	50+100，边之间都没有相同的顶点
7	1+100，最少边最多点
8	17+100，随机生成的较大样例，稀疏图
9	100+100，最大样例，环状图

Input samples can be found in the 'sample' folder you download.

Sample Output

No.	Output
1	12 002 004 006 008 009 014 015 016 017 018 019 020 239
2	1 002 10
3	1 001 20
4	5 002 004 005 006 008 73
5	3 001 002 003 37
6	Time Limit Exceeded. This might be one of the worst cases and is hard to optimize. Even by the formal B-K algorithm it takes a rather long time.
7	99 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 4706
8	89 003 004 006 007 008 011 012 013 014 015 016 018 019 020 021 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059 060 061 062 063 064 066 067 068 069 070 071 072 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 4014
9	50 002 004 006 008 010 012 014 016 018 020 022 024 026 028 030 032 034 036 038 040 042 044 046 048 050 052 054 056 058 060 062 064 066 068 070 072 074 076 078 080 082 084 086 088 090 092 094 096 098 100 2219

Analysis and Comments

First we restate the problem through graph theory.

The input data represents a graph $G(V, E)$. Each fruit available is a vertex in it, and each pair of unsafe fruits represents is line connecting the vertices of the two fruits. We are asked to find a subgraph of G so that there're no edges in it (also the price should be lowest). N, M, respectively, are the total number of vertices and edges.

We call a vertex in the graph an **independent vertex** if it's not connected to any other vertex in the graph, and the set of these vertices **independent set**. Then the problem require us to find the **Maximum independent set** of graph G. BTW, this is equivalent to find the **Maximum complete subgraph** of the complement graph.

Time Complexity:

- **Data read-in and sorting**: Reading each tip and price information takes $O(1)$ time, amounting to $O(\max\{N, M\})$ in total。The time complexity of our sorting algorithm is $O(M^2)$. So finally the time complexity is $O(M^2)$ (Note that $N \leq M(M-1)/2 \sim O(M^2)$) 。
- **Backtracking**: If the number of edges N is not taken into consideration, apparently the size of the searching tree is $O(2^N)$ before pruning. But each effective edge can reduce the size of the solution space by a certain scale, and this is up to the size and shape of the connected set in which the two vertices it connects lay.

  Larger the connected set is, smaller the scale of reduction is. That's because some possible cases have already been excluded by the connected set. For example, consider a graph with three vertices, and there's no edge between $p1, p2$, then adding the edge $(p2, p3)$ reduces the size of the searching tree by $1/4$ (1×2 cases). However, if $p1, p2$ is connected, then the number will only be $1/6$ (1×1 cases). Here we use proportion because we want to keep it right if there's more vertices independent of these three vertices.
  It's known that a new edge reduces the size by $1/4$ at most, in which case the two vertices it connects are both independent. Then the lower bound of the size is $O((\frac{3}{4})^N \cdot 2^M)$ for a graph with N edges and M vertices (Sample 6). In worse case, such as a 'monocentric flower-like graph', it can be calculated the the size is $O((1 - \frac{2 - 2^{1-N}}{M}) \cdot 2^M) \sim O(2^M)$.

  Nevertheless, this's not the worst case.

If the graph is complete, the size would be $O(M)$; or if there's no edge, the size reaches the upper bound $O(2^M)$.
The effect of pruning in our program is hard to estimated. A similar popular algorithm used exactly for this problem has proven, in this paper, to have the worst time complexity of $O(3^{\frac{M}{3}})$, which is $O(2^{\frac{\ln 3}{3}M})$. Therefore, it's reasonable to estimate the worse time complexity of our algorithm as $O(2^{KM})$, where K is between $\frac{\ln 3}{3}$ and 1.

- **Print-out**: With the number of fruits printed no more than M, we have upper bound $O(M)$ 。

Summing up, we have a time complexity of $O(2^{KM})$, which means most of the time is consumed in the backtracking step.

Space complexity

- **水果信息储存**: struct `fruit_array` contains an array of size $O(M)$, so the space for the two baskets is $O(M)$, and the array `fruits` has $MAXID$ elements where $MAXID \geq M$, giving the space complexity $o(M^2)$。In most cases $MAXID \sim M$, hench the complexity is $\Theta(M^2)$ 。
- **编号-ID值转换数组**: `ind` has M elements, with complexity$O(M)$。

- **回溯算法中使用的信息**：`mark` and `fruits` has $o(M)$ elements（in most cases$O(M)$），the size of `dp` equals to the maximum recursion depth of *dfs* function which is no more than M, hench total space complexity $O(M)$。

Summing up, we have a space complexity of $\Theta(M^2)$（or more strictly, $o(M^2)$）.

Result Analysis

We can use the macro definition `#define DEBUG` to print the running time of the *dfs* module out to the console, and `#define FASTER_ALGORITHM` will optimize the searching process by dp.

The output results of sample 1, solved through these two algorithms are as follows.

```
NAIVE DFS: 0 secs
12
002 004 006 008 009 014 015 016 017 018 019 020
239
FASTER with DP: 0 secs
12
002 004 006 008 009 014 015 016 017 018 019 020
239
```

Running results of sample 1, the left through naive dfs while the right through dfs optimized by dp

Both answers are correct and both take such a short time that make no difference here. But their perfomances will differ greatly in large samples, as shown below

```
NAIVE DFS: 36.164 secs
89
003 004 006 007 008 011 012 013 014 015 016 018 019 020 021 025 026 027 028 029 030 031 032 033 034
035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059
060 061 062 063 064 066 067 068 069 070 071 072 074 075 076 077 078 079 080 081 082 083 084 085 086
087 088 089 090 091 092 093 094 095 096 097 098 099 100
4014
FASTER with DP: 0.252 secs
89
003 004 006 007 008 011 012 013 014 015 016 018 019 020 021 025 026 027 028 029 030 031 032 033 034
035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059
060 061 062 063 064 066 067 068 069 070 071 072 074 075 076 077 078 079 080 081 082 083 084 085 086
087 088 089 090 091 092 093 094 095 096 097 098 099 100
4014
```

Running results of sample 8, a sparse graph

We can see that dfs takes 36 seconds before optimizing and less than 0.3 second after, which shows the dp optimization do make great progressions. The sample describes a sparse graph with 100 vertices and 17 edges. There are few restrictions compared to the number of points in this case, leading to a huge search tree. The naive dfs will have to search through a large number of nodes, which takes a long time.

While for the optimized dfs, by the first few iterations, the number in `dp` array will soon reach its maximum in theory (for example, `dp[97]` reaches its theoretical maximum 4). Therefore, in subsequent iterations, lots of subtrees of large sizes are pruned, with a few long paths left, cutting the solving time down.

```
098 83
099 30
100 42

FASTER with DP: 0 secs
50
002 004 006 008 010 012 014 016 018 020 022 024 026 028 030 032 034 036 038 040 042 044 046 048 050
052 054 056 058 060 062 064 066 068 070 072 074 076 078 080 082 084 086 088 090 092 094 096 098 100
2219
```

Running results of sample 9, a ringlike graph of maximum size,
not solved after naive dfs running for 10 minutes

As for example 9, a ringlike graph, and again a sparse graph, the dp-optimized dfs takes even a shorter time, mostly because of the increasing restrictions. But it's interesting that the naive dfs isn't able to find out the solution after running for 10 minutes. A possible reason might be that this dfs, in fact, also deploys a pruning strategy, that consider the number of fruits left to be chosen (which is less efficient than the dp one because it assumes that all fruits left are picked up into the basket.) In sample 8, the local optimal solution has already been well enough at first to prune most of the large subtrees, while sample 9 restricts the local optimal solution further, so large subtrees are not pruned and takes a long time to search through.

What's worse, both algorithms, even the popular B-K algorithm, takes an extremely long time (more than 30 minutes!) to find a solution for sample 6, in which 100 vertices are connected and only connected to one other vertex. We also attribute this to the large subtrees that cannot be pruned even when using the `dp` array.

心得体会：

The problem is not complicated, as stated above, because it equals to finding the maximum independent subgraph. However, it's an NP problem that cannot be solved in polynomial time. Naive dfs will exceed time limit under the large-input test case in PTA.

So the crucial key here, for a problem that is bound to take exponential time, is to find an effective pruning strategy, cutting down the size of the solution space. But this is not easy since it requires keen insight and comprehensive thinking. For example, in this project, we might take it for granted that calling dfs multiple times in a loop will take a longer time than to call it only once. However, in the former way, we can store precedent information in an array, allowing us to prune trees more effectively and skip most of the repeated work. A more subtle estimation shall make such a great improvements in running time!

But on the other side, on matter how we improve our algorithm, the time complexity is still at exponential level. So elaborate input data that avoids the program to prune large subtrees can result in bad performance of the algorithm. In fact, sample 6 is unintentionally found when constructing input sample. So we should construct data with different features when measuring the performance of an algorithm so as to get a more comprehensive testing result.

Appendix: Source Code

main.cpp

```
#include<stdio.h>
#include<string.h>
#include<time.h>

#define MAXN 102
#define MAXID 1002

#define FASTER_VERSION
//#define DEBUG

struct fruit_array{
    int p, value;//p是fruit的大小，value是水果总价（篮子）/或/价格（水果）
    int fruit[MAXN];//篮子中的所有水果（篮子）/或/排斥的所有水果（水果）
} bestBasket, curBasket, fruits[MAXID];
```

```
int K, N;//tips数和水果数
int mark[MAXID]={0};//当前篮子中与其排斥的水果种数
int dp[MAXN] = {0};//从编号i~N的水果中满足条件的组合中水果种数的最大值
int ind[MAXN];//将编号转换为ID

/*
    作用：对传入数组进行选择排序，从小到大排序
    参数：a-数组头地址；n-数组长度
    返回值：无
*/
void min_sort(int *a, int n){
    for(int i = 0; i < n; i++){
        int mini = i;//a[i]~a[n]中的最小值
        for(int j = i + 1; j < n; j++){//寻找最小值
            if(a[j] < a[mini]) mini = j;
        }
        int temp = a[i]; a[i] = a[mini]; a[mini] = temp;//放置最小值
    }
}

/*
    作用：搜索从编号为round~N的水果中能选择的最大种数
    参数：round-水果编号最小值
    返回值：无
*/
void dfs(int round) {
    for (int i = round; i <= N + 1; i++) {//遍历每个可能可以加入篮子的水果

        if (i == N + 1) {//递归边界条件，说明篮子已填充完毕，没有可以新增的水果
            if (curBasket.p > bestBasket.p || (curBasket.p == bestBasket.p &&
curBasket.value < bestBasket.value))//当前篮子比局部最优篮子要优
                memcpy(&bestBasket, &curBasket, sizeof(struct fruit_array)); //更
新局部最优篮子
            break;//停止递归
        }

        int ii = ind[i];//获取水果ID

        if (N - i + 1 < bestBasket.p - curBasket.p //粗糙剪枝条件
#ifdef FASTER_VERSION
            || dp[i+1] + 1 < bestBasket.p - curBasket.p //精细剪枝条件
#endif
            ) break;//不论后面怎么选，都不可能出现比先前的局部最优更优的结果，直接跳过整棵搜
索树
        if (mark[ii] > 0 || fruits[ii].value <= 0) continue;//该水果和已有水果排斥或
不可选（没有价格信息）

        //水果可以加到篮子里，更新篮子信息与排斥信息
        curBasket.fruit[curBasket.p++] = ii;
        curBasket.value += fruits[ii].value;
        for(int it = 0; it < fruits[ii].p; it++) mark[fruits[ii].fruit[it]] +=
1;

        dfs(i + 1);//水果添加完毕，递归搜索子树

        //回溯，水果从篮子里拿出，undo篮子信息与排斥信息
        for(int it = 0; it < fruits[ii].p; it++) mark[fruits[ii].fruit[it]] -=
1;
```

```
            curBasket.value -= fruits[ii].value;
            curBasket.p--;
        }
    }

    int main() {
        bestBasket.value = 1e8;

        scanf("%d %d", &K, &N);
        int i1, i2;

        //读入排斥信息
        for (int i = 0; i < K; i++) {
            scanf("%d %d",&i1,&i2);
            if(i1 > i2) fruits[i2].fruit[fruits[i2].p++] = i1;/*仅需要将大编号水果放到*/
            else fruits[i1].fruit[fruits[i1].p++] = i2;       /*小编号水果的排斥列表里*/
        }

        //读入水果价格信息
        for (int i = 0; i < N; i++) {
            scanf("%d %d",&i1,&i2);
            fruits[i1].value = i2;
            ind[i+1] = i1;//储存编号-ID转换信息
        }
        min_sort(ind+1, N);//按ID从小到大排序

        clock_t start, end;//计时变量
        start = clock();//计时开始
    #ifdef FASTER_VERSION
        for(int i = N; i > 0; i--){//dp思想的dfs（类似B-K算法）
            dfs(i);
            dp[i] = bestBasket.p;//储存i~N最优解
        }
    #else
        dfs(1);//朴素dfs
    #endif
        end = clock();//计时结束
    #ifdef DEBUG
        printf("%g secs\n", (double)(end-start)/CLOCKS_PER_SEC);//计算耗费时间
    #endif

        //输出结果
        printf("%d\n", bestBasket.p);
        for (int i = 0; i < bestBasket.p; i++) printf("%s%03d", i == 0 ? "" : " ",
    bestBasket.fruit[i]);
        printf("\n%d\n", bestBasket.value);
    }
```

Declaration

We hereby declare that all the work done in this project titled "Project 2 - Safe Fruit - Report" is of our team's independent effort.