Project 2 Safe Fruit

Group 24

Date: 2020-04-12

Project 2 Safe Fruit

1. Introduction

1.1 Description

We are given tips about some pairs of fruits which must not be eaten at the same time, together with a basket of fruits. It is our goal to find out the maximum number of safe fruits(the fruits that we can eat at the same time), with its total price. If there're more than one solution to the maximum number, we're supposed to output the one with the lowest total price.

We'll use backtracking to solve this problem.

2. Algorithm Specification

2.1 Data Structure Specification

2.1.1 tips[ ] array and tipNum[ ] array

- tips[ ]

We use a pointer array to store all the input tips, declared as `int* tips[MAX]`. Each element of this array is a pointer points to an array that stores all the fruits which cannot be eaten with one specific fruit. Take `tips[i]` as an example, it points to an array whose elements cannot be eaten with fruit i.

Note that each time we input an unsafe fruit with fruit i, we would use the function `realloc()` to rearrange the space of array `tips[i]`, which help us to minimize the space.

- tipNum[ ]

It stores the number of unsafe fruits associated with each kinds of fruits. That is, `tipNum[i]` is equal to the number of fruits cannot be eaten with fruit i, which is just the number of elements of `tips[i]`.

2.2 Algorithm Specification

2.2.1 Main Program

> Pseudocode for main()

```
int main()
{
    input N and M
    /*N:the number of tips; M:the kinds of fruits*/

    input and initialization
    /*initialize the tips[], tipNum[], price[] and index[]*/

    execute BackTracking
    /*using backtracking to find the optimal solution*/

    output the result
}
```

2.2.2 Main Algorithms

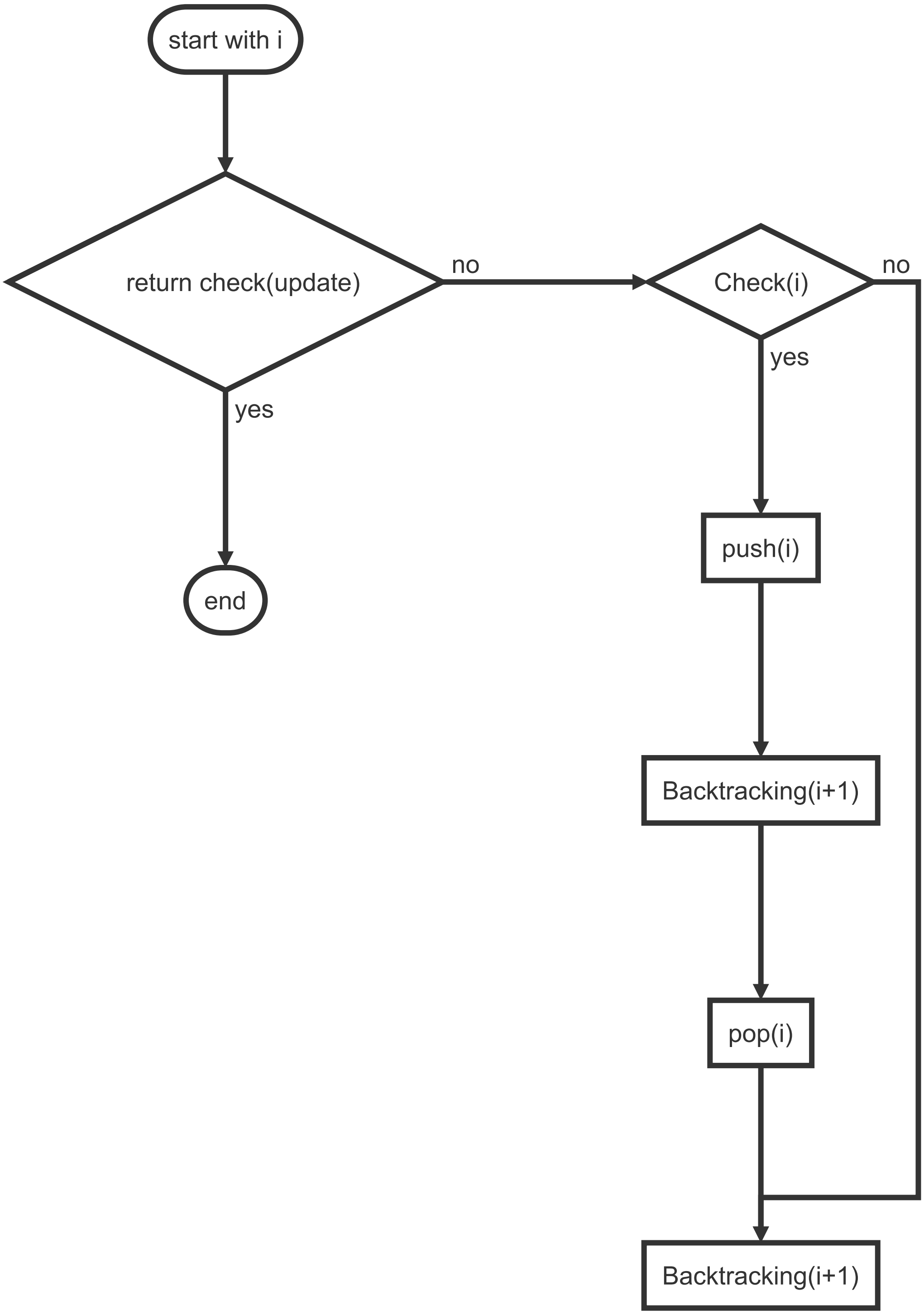
> Pseudocode for BackTracking()
>
> Notes: array `res[]` stores the final results while `currRes[]` stores the current optimal result during the search process.

```
void BackTracking(int k)
{
    if(fits pruning condition)
        return;

    if(k==M){
        if(need to update)
            update res[]
        return;
    }

    OK = check if k can be eaten with currRes[]
    if(OK){
        Count k in
        BackTracking(k+1)
        Undo(k)
    }
    BackTracking(k+1)
}
```

The flow chart is as follows



3. Testing Results

- **Sample**

  input:

  > 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

output：

> 12
>
> 002 004 006 008 009 014 015 016 017 018 019 020
>
> 239

- Different solutions

  input:

  > 3 7
  >
  > 001 002 003 004 005 006
  >
  > 001 1 002 2 003 3 004 4 005 5 006 6 007 7

  output:

  > 4
  >
  > 001 003 005 007
  >
  > 16

- Min M

input:

> 0 0

output:

>

- Max M

  The code generating the test case included in Appendix

  input:

  > 4 100
  >
  > 075 024 030 077
  > 059 096
  > 084 021
  >
  > 001 668
  > 002 851
  > 003 283
  > 004 713
  > 005 400
  > 006 38
  > 007 826
  > 008 119
  > 009 590
  > 010 82
  > 011 282
  > 012 798
  > 013 459
  > 014 555
  > 015 215
  > 016 902
  > 017 688
  > 018 770
  > 019 558
  > 020 112
  > 021 748
  > 022 650
  > 023 552
  > 024 193
  > 025 71
  > 026 826
  > 027 255
  > 028 677
  > 029 904
  > 030 907
  > 031 281
  > 032 554
  > 033 842
  > 034 251
  > 035 115
  > 036 519
  > 037 426

```
038 353
039 296
040 820
041 76
042 386
043 714
044 323
045 454
046 874
047 624
048 384
049 573
050 727
051 33
052 358
053 959
054 507
055 647
056 202
057 633
058 368
059 583
060 851
061 178
062 417
063 49
064 338
065 231
066 470
067 11
068 110
069 384
070 152
071 779
072 163
073 219
074 198
075 804
076 788
077 592
078 368
079 800
080 487
081 693
082 850
083 391
084 337
085 50
086 947
087 551
088 553
089 312
```

```
090 214
091 45
092 17
093 837
094 616
095 971
096 826
097 467
098 196
099 866
100 615
```

output:

```
96
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020
022 023 024 025 026 027 028 029 031 032 033 034 035 036 037 038 039 040 041 042
043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059 060 061 062
063 064 065 066 067 068 069 070 071 072 073 074 076 077 078 079 080 081 082 083
084 085 086 087 088 089 090 091 092 093 094 095 097 098 099 100
45032
```

4. Analysis and Comments

4.1 Time Complexity

If we don't take the pruning into consideration, the time complexity is the sum of all the paths in the game tree(the game tree is constructed according to whether we take one fruit). So the time complexity is $O(2^N)$.

4.2 Space Complexity

The space complexity has a lot to do with the input, which varies in the density of the unsafe fruits.

In the worst case, all fruits are unsafe with each other, the space complexity of all the arrays of `tips[]` takes $O(N^2)$. The backtracking algorithm takes $O(N)$ (the maximum height of the game tree). As a result, the space complexity is $O(N^2)$ in the worst case.

4.3 Comments

- The first problem we have to solve is to decide how to store the input data. At first thought, *Union Find* would be a good choice. But when we look into it, it comes to us that *Union Find* is only suitable for *equivalence relations*, and the fruit taboo is clearly not an equivalence relation(not transitive). So we'd better find another solution. For the convenience, we used a pointer array to store the tips. (Actually, using a `int**` pointer and malloc the space for the tips[ ] would take less space)
- We used two arrays `res[]` and `currRes[]`, to store the final result and current result respectively. Basically, we build the game tree according to whether we take the current fruit into the basket(push it into `currRes[]`). And when we reach a leaf of the game tree, we compare the number of fruits and costs between `res[]` and `currRes[]` to decide if we need to update the `res[]`. At each recursion, we use the `check()` function to examine whether the current fruit i can be eaten with those in the `currRes[]`, if it's safe, push it into `currRes[]` and go to the next level, when get back, pop it and go to next level(cover the

condition that we don't take it); otherwise, we simply move on to the next condition(don't take it).

5. Author List

Programmer: WangRui

Tester: LiYalin

Writer: WangRui / OuyangHaodong

6. Declaration

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

7. Appendix

7.1 Source code in C

> You can open the code.c in IDE for a better view

```c
#include<stdio.h>
#include<stdlib.h>

#define MAX 101

typedef enum { false, true }bool;

/*********************************************************************
                          Global Variables

@ int* tips[]: An array used to store the pointers which
        point to the tip array of each fruit. For example,
        tips[i] points to an array whose elements cannot
        be eaten with i. We use the function "realloc" to
        minimize the space.

@ tipNum[]: The number of unsafe fruits associated with each fruit.
        That is, tipNum[i] is equal to the number of elements
        in tips[i].

@ price[]: An array used to store the price of fruit. The
        index of this array is fruit id.

@ index[]: An array used to convert the index(range from 0 to M-1)
        to the fruit ID. (since the fruit ID may not start from 0 and
        be continuous, we need this array for index conversion)

@ res[]: An array used to store the final result(the maximum fruit
        which can be eaten at same time).

@ resPointer: an integer which is the pointer of the array res[]

@ currRes[]: An array used to store the current result during the
        backtracking. We can copy the elements in the currRes[]
        to the res[] to update the search result. ( currRes[] stores
        the fruit we currently decide to eat)

@ currResPointer: an integer which is the pointer of the array
        currRes[].

@ cost: the final total cost for the maximum fruit.(those in the res[])

@ currCost: the current total cost for the current fruits. (those in
        the currRes[]).

@ M: the number of fruits in the basket.

@ N: the number of tips.

*************************************************************************/

int* tips[MAX];
int price[MAX], index[MAX], tipNum[MAX];
int res[MAX], currRes[MAX];
int resPointer = -1, currResPointer = -1;
int cost, currCost;
int M, N;

/*********************************************************************
                          Functions

@ initialize(): A function used to store the input tips and the prices
        of each fruit.

@ BackTracking(): Using backtracking to find the maximum fruits.

@ check(): check if the current fruit can be eaten with the fruits
        we have searched so far (those in the currRes[])

@ output(): Output the result in the required format.

@ quickSort(): Using quicksort to sort the res[] into ascending order.

@ partition(): the partition function used in the quickSort().

@ swap(): swap the position of two values in an array.

*************************************************************************/


void initialize(void);
void BackTracking(int index);
bool check(int index);
void output(void);
void quickSort(int array[], int low, int high);
int partition(int array[], int low, int high);
void swap(int* a, int* b);

int main()
```

```
{
    scanf("%d %d", &N, &M);

    initialize();

    BackTracking(0);

    output();

    return 0;

}

void initialize(void)
{
    int i;
    int id1, id2;
    int id, money;

    for (i = 0; i < MAX; i++) // first initialize the tipNum[] to 0
        tipNum[MAX] = 0;

    for (i = 0; i < N; i++) {
        scanf("%d %d", &id1, &id2); // input 2 unsafe fruits id1 and id2
        tips[id1] = realloc(tips[id1], sizeof(int) * (tipNum[id1] + 1)); // increase the space of array tips[id1] by 1
        tips[id2] = realloc(tips[id2], sizeof(int) * (tipNum[id2] + 1)); // increase the space of array tips[id2] by 1
        tips[id1][tipNum[id1]++] = id2; // store the tip id1 cannot be eaten with id2 to the both array:tips[id1] and tips[id2]
        tips[id2][tipNum[id2]++] = id1;
    }

    for (i = 0; i < M; i++) {
        scanf("%d %d", &id, &money);
        price[id] = money; // store the price of each fruit(the index is ID)
        index[i] = id;
        // the ID of each fruit is indexed from 0 t0 M-1 (since the fruit ID may not start from 0 and be continuous, we need this array for index conversion)
    }
}


void BackTracking(int k)
{
    int id, i;
    bool OK;

    // even if we search all the unsearched fruit, the number of fruits qualified is still less than those in the res[], pruning
    if (resPointer > currResPointer + M - k)
        return;

    if (k == M) { // we have reached the leaf of the game tree
        if ((resPointer < currResPointer) || (resPointer == currResPointer && cost > currCost)) { // check if we need to update the res[] and cost
```

```c
            for (i = 0; i <= currResPointer; i++)
                res[i] = currRes[i];
            resPointer = currResPointer;
            cost = currCost;
        }
        return;
    }
    OK = check(k); // check if kth fruit can be eaten with those in the currRes[]
    id = index[k]; // get the id of the kth fruit
    if (OK) {      // if it is safe, push it into currRes[], and search the next fruit
        currRes[++currResPointer] = id;
        currCost += price[id];  // update the price
        BackTracking(k + 1);
        currCost -= price[id];  // pop the kth fruit from currRes[](we don't eat it), and search the next fruit
        currResPointer--;
    }
    BackTracking(k + 1);
}

bool check(int k)
{
    int id = index[k], i, j;
    bool flag = 0;

    for (i = 0; i <= currResPointer; i++) { // for each fruit in currRes[] (the fruit we currently decide to eat)
        for(j=0;j<tipNum[id]; j++)  // check all the fruits that cannot be eaten with kth fruit
            if (tips[id][j] == currRes[i]) { // the kth fruit cannot be eaten with the fruits in the currRes[]
                flag = 1;
                break;
            }
    }
    if (flag)
        return false;
    return true;

}

void output(void)
{
    if (M == 0)     // boundary case
        return;
    int i;

    quickSort(res, 0, resPointer);

    printf("%d\n%03d", resPointer + 1, res[0]);
    for (i = 1; i <= resPointer; i++)
        printf(" %03d", res[i]);
    printf("\n%d\n", cost);
}

void quickSort(int array[], int low, int high)
```

```
{
    if (low < high) {
        int pi = partition(array, low, high);
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
}

int partition(int array[], int low, int high)
{
    int pivot = array[high];  // always set the last element as the pivot.
    int i = low - 1, j;

    for (j = low; j <= high - 1; j++) { // low-i: smaller than pivot; i-j: bigger than pivot; j-high: to be examined
        if (array[j] < pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]);
    return (i + 1);
}

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

7.2 Test code for generate Max N in C++

> It may generate case whose solutions are not unique, but it doesn't matter for test

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <time.h>
using namespace std;

#define N 100
#define M 4
int main() {
    /**********************************************************************
                          Global Variables

    @vector<int> price: An array used to store the price of fruit. The
        index of this array is index from 0 ~ 99.

    @vector<map<int,int>> tips: An array used to store the tips that the two
        fruits listed can't be eaten together. The index of each pair is
        fruit id.

```

```cpp
*************************************************************************/
    vector<int> price(N);
    vector<pair<int, int>> tips;

    srand(time(NULL));
    /*********************************************************************
                          Generate prices

*************************************************************************/
    for (int i = 0; i < N; i++) {
        price[i] = rand() % 1000 + 1;
    }

    /*********************************************************************
                          Generate tips

*************************************************************************/
    for (int i = 0; i < M; i++) {
        int fruit1 = rand() % N + 1;
        int fruit2 = rand() % N + 1;
        if (fruit1!=fruit2\
            &&(find(tips.begin(), tips.end(), pair<int,int>(fruit1,fruit2)))==tips.end()\
            &&(find(tips.begin(),tips.end(),pair<int,int>(fruit2,fruit1)))==tips.end()) {
            tips.push_back(pair<int, int>(fruit1, fruit2));
        }
    }

    /*********************************************************************
                          Output
    @cout<<setw(3)<<setfill('0'): ensure the format of fruit id

*************************************************************************/
    cout << tips.size() << " " << price.size() << endl;

    for (int i=0;i<tips.size();i++)
        cout << setw(3) << setfill('0') << tips[i].first << " " << setw(3) << setfill('0') << tips[i].second << endl;

    for (int i = 0; i < price.size(); i++)
        cout << setw(3) << setfill('0') << i+1 << " " << price[i] << endl;

    system("pause");
}
```