Safe Fruit

Group 1 Wang Kedi, Zhao Yilei, Shen Yunfeng

Date: 2022/03/31

Chapter 1: Introduction

In daily life, consuming certain fruits together may lead to disastrous consequences. For example, eating banana with cantaloupe could cause kidney deficiency. In this situation, how to select safe fruit for eating has become a critical problem. This project, we should present the safe fruit group, which covers largest number of safe fruit and costs least among the solutions covering the same number of fruits.

To handle this problem, we need to scan a list of N tips and M fruits, and then output the largest collection of safe fruits in quantity with the minimum total price. During this process, we use backtracking algorithm to enumerate all possibilities at first, unfortunately, it takes too long.

We optimize it with Bron-Kerbosch Algorithm, which accelerated the process of searching the maximum clique. Although other algorithms for solving the max clique problem have theoretically shorter running time when inputs has few maximal independent sets, BK algorithm and its improvements are regularly reported to be more efficient.

Chapter 2: Data Structure / Algorithm Specification

Data Structure:

Adjacency matrix —— implemented by two dimensional array

Algorithm Specification:it

Bron-Kerbosch Algorithm

Specification:

A **clique** in a graph G is a subset of the vertices such that every two vertices in it are the two endpoints of an edge in G. A **maximal clique** is a clique to which no more vertices can be added.

In order to find the maximal clique in an undirected graph G, we first divide all the vertexes into three disjoint groups: R, P, X, among which R & X are initially empty and P included every vertexes at the very beginning. And in the end, we output R as the maximum clique in graph G.

In recursive process, the algorithm enumerate every vertex in P in order, if P & X are both empty, then R is a maximal clique, else if only P is empty, we should backtrack. For each vertex in P, recursive call happens when new vertex V is added to R ,at the same time P and X are restricted to the neighbor set of V, which finds and reports all clique extensions of R that contain V. Then, move V from P to X to exclude it from consideration in future cliques and continues with the next vertex in P.

In the Safe Fruit Problem, if we consider two fruits in given tips are connected, then we are supposed to output the maximal independent set on original graph G, which equals to the maximal clique in its complementary graph. Thus consider those vertex pairs in tips are not connected, and every other vertex pairs are connected, the answer equals to the maximal clique in such a graph.

Pseudo Code:

```
bool BK(bool R[], bool P[], bool X[])
{
```

```
    if(P[] & X[] are empty)
    {
        searching finished, check whether current solution is optimal;
        if(yes)
        {
            update min price, max fruit number, final solution;
            return true;
        }
        else    return false;
    }
    for(each vertex v in P[])
    {
        BK(R[]U{v}, P[]∩{all vertexes adjacent to v}, X[]∩{all vertexes
adjacent to v}));
        exclude {v} form P[];
        insert {v} into X{};
    }
}
```

Check & Update

Specification:

After found a feasible solution, we need to check whether the the current solution is locally optimal before updating final solution. We can discuss it in three scenarios:

1. If fruits covered in current solution were fewer than locally optimal one, we just eliminate this solution.
2. If fruits covered in current solution were more than locally optimal one, we should update the min price, max fruit number, and final solution.
3. If fruits covered in current solution were equal to locally optimal one, we should calculate the total cost of current solution at first.

   3.1 If current cost < min cost, update the min price, max fruit number, and final solution.

   3.2 If current cost > min cost, we just eliminate this solution.

   As the solution is guaranteed to be unique, there is no equivalent case.

Pseudo Code:

```
bool Check()
{
    if(current fruit number < max fruit number) return false;
    else if(current fruit number > max fruit number)
    {
        calculate the total price;
        update min price, final solution, max fruit number;
        return true;
    }
    else if(current fruit number = max fruit number)
    {
        calculate the total price;
        if(current total price < min price)
```

```
        {
            update min price, final solution;
            return true;
        }
        else return false;
    }
}
```

Chapter 3: Testing Result

Sample 1

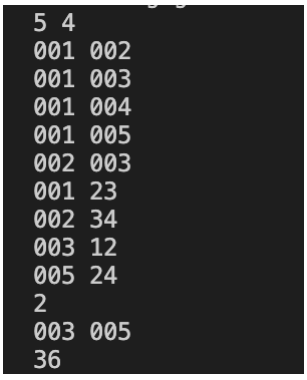
Purpose:

In this sample, we take into consideration of some special case. We need to judge whether the maximum plan of these fruits has some kind of fruit free, which is not legal. So in this sample, though 003 and 004 will have much smaller price, 004 is forbidden to pick up.

Input:

```
5 4
001 002
001 003
001 004
001 005
002 003
001 23
002 34
003 12
005 24
```

Output:




```
2
003 005
36
```

Status: pass

Sample 2

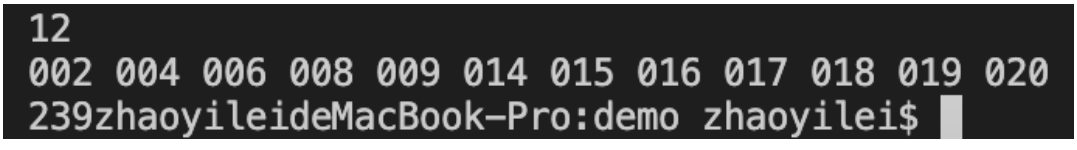
Purpose:

This sample is given by the project. It has no special cases and has the decent kinds of fruits.

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
```

Sample 2 Output



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

Status: pass

Sample 3

Purpose:

This sample is much bigger than the former ones. We will need to deal with 60 fruits. If we use DFS algorithm, it will be much harder to find the right answer.

Input:

```
60 60
011 027
044 052
003 042
015 012
027 051
044 051
049 036
012 039
036 048
028 055
032 026
012 034
031 037
010 029
058 055
026 038
052 003
002 011
045 026
006 012
016 045
013 037
039 030
054 004
034 046
008 049
035 048
026 035
040 020
053 054
001 002
019 009
060 012
024 001
030 033
019 059
```

```
039 036
007 059
017 060
011 009
045 009
015 041
039 055
048 037
036 053
005 034
004 045
051 047
025 057
058 044
032 001
037 019
050 024
003 023
015 005
039 010
034 031
002 045
042 038
047 021
001 7
002 10
003 3
004 4
005 10
006 10
007 7
008 6
009 3
010 9
011 7
012 1
013 3
014 1
015 7
016 6
017 10
018 3
019 5
020 3
021 4
022 5
023 10
024 10
025 6
026 4
027 8
028 1
```

```
029 5
030 7
031 8
032 10
033 1
034 4
035 9
036 7
037 8
038 2
039 8
040 9
041 4
042 9
043 9
044 7
045 3
046 3
047 6
048 5
049 1
050 2
051 5
052 5
053 8
054 4
055 5
056 8
057 1
058 3
059 7
060 8
```

Output:

We can get the answer in a short time.

```
055 5
056 8
057 1
058 3
059 7
060 8
35
001 004 005 006 007 011 013 014 016 018 019 020 021 022 023 026 028 029 031 033 039 04
1 042 043 046 048 049 050 051 052 053 056 057 058 060
zhaoyileideMacBook-Pro:demo zhaoyilei$
```

Status: pass

Chapter 4: Analysis and Comments

Time Complexity:

According to Moon & Moser's research[2], any n-vertex graph has at most $3^{n/3}$ maximal cliques, and the worst-case of the Bron–Kerbosch algorithm matches this bound, and thus its time complexity is $O(3^{n/3})$.

Space Complexity:

In this project, it tool O(1) to store the index, price, and the connectivity of given fruits,.

In the worst case of Bron–Kerbosch algorithm, every node would be enumerated in the backtracking, which would took $O(n^2)$ auxiliary space.

To sum up, the total space complexity of this project is $O(1) + O(n^2) = O(n^2)$.

Appendix: Source Code

```
#include <stdio.h>
#include <stdlib.h>
#define INF 10000000

typedef enum{false, true} bool;
int ref[1001];
int fruitCount;
int mat[1001][1001];
int price[1001];
bool final[1001];
int minPrice, maxFruitCount;
int countR, countP, countX;
bool BK(bool R[], bool P[], bool X[]);
bool check(bool R[]);
int main()
{
    minPrice = INF;
    int relationCount;
    scanf("%d%d", &relationCount, &fruitCount);
    countP = fruitCount;
    int i, f1, f2;
    int count = 0;
    for(i = 0; i < relationCount; i++)
    {
        scanf("%d%d", &f1, &f2);
        if(ref[f1] == 0)    ref[f1] = ++count;
        if(ref[f2] == 0)    ref[f2] = ++count;
        if(!mat[ref[f1]][ref[f2]])  mat[ref[f1]][ref[f2]] = mat[ref[f2]]
[ref[f1]] = 1;
```

```
    }
    bool R[101], P[101], X[101];
    for(i = 1; i <= fruitCount; i++)
    {
        scanf("%d%d", &f1, &f2);
        if(ref[f1] == 0)    ref[f1] = ++count;
        price[ref[f1]] = f2;
        R[i] = X[i] = false;
        P[i] = true;
    }
    BK(R,P,X);
    printf("%d\n", maxFruitCount);
    bool hasPrint = false;
    for(i = 0; i < 1000; i++)
    {
        if(!ref[i]) continue;
        if(!final[ref[i]])    continue;
        if(hasPrint)    printf(" ");
        else hasPrint = true;
        if(final[ref[i]])   printf("%03d", i);
    }
    if(maxFruitCount != 0) printf("\n");
    printf("%d", minPrice);
    return 0;

}
bool BK(bool R[], bool P[], bool X[])
{
    bool find, judge;
    if(!countP && !countX)
    {
        find = check(R);
        return find;
    }
    int saveR, saveP, saveX;
    saveR = countR;
    saveP = countP;
    saveX = countX;
    bool nR[101], nP[101], nX[101];
    int i, j;
    for(i = 1; i <= fruitCount; i++)    //for each fruit in P
    {
        if(countR + countP < maxFruitCount)  return find;
 //impossible to get more fruit
        if(!P[i])   continue;
        for(j = 1; j <= fruitCount; j++)    //update sets
        {
            nR[j] = R[j];
            nP[j] = P[j];
            nX[j] = X[j];
            if(j == i)
            {
```

```
                if(nP[j])
                    nP[j] = false, countP--;
                if(nX[j])
                    nX[j] = false, countX--;
                if(!nR[j])  //add i into R to test whether the max
solution should have i
                    nR[j] = true, countR++;
            }
            //nP = P ∩ N{i}, check whether fruit[j] is N{i}
            if(nP[j] && mat[i][j])  //j in P but collision(not neighbor)
            {
                nP[j] = false;
                countP--;
            }
            if(nX[j] && mat[i][j])
            {
                nX[j] = false;
                countX--;
            }
        }
        judge = BK(nR, nP, nX);
        if(!find)   find = judge;
        //update P and X
        countP = saveP;
        countX = saveX;
        countR = saveR;
        if(P[i])    //P = P - {v}
            P[i] = false, countP--, saveP = countP;
        if(!X[i])   //X = X∪{v}
            X[i] = true, countX++, saveX = countX;
    }
    return find;
}
bool check(bool R[])
{
    int i;
    if(countR > maxFruitCount)  //more fruit
    {
        minPrice = 0;
        for(i = 1; i <= fruitCount; i++)
        {
            if(R[i])    minPrice += price[i];
            final[i] = R[i];
        }
        maxFruitCount = countR;
        return true;
    }
    else if(countR < maxFruitCount)
        return false;
    else if(countR == maxFruitCount)
    {
        int p = 0;
```

```
        for(i = 1; i <= fruitCount; i++)
        {
            if(R[i])    p+=price[i];
        }
        if(p >= minPrice)   return false;
        else
        {
            minPrice = p;
            for(i = 1; i <= fruitCount; i++)
                final[i] = R[i];
            return true;
        }
    }
}
```

References:

[1] Wikipedia, 'Bron–Kerbosch algorithm', https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm

[2] Moon, J. W.; Moser, L. (1965), "On cliques in graphs", Israel Journal of Mathematics, 3: 23–28, doi:10.1007/BF02760024, MR 0182577.

Author List:

Name	Work
Wang Kedi	Code
Zhao Yilei	Debug & Report Chap 3
Shen Yunfeng	Debug & Report Chap 1,2,4

Declaration:

We hereby declare that all the work done in this project titled "pj3" is of our independent effort as a group.

Signatures:

Each author must sign his/her name here.