Safe Fruit

Author：ommited

Date：2020-04-05

Chapter 1: Introduction

There are a lot of tips telling us that some fruits must not be eaten with some other fruits, or we might get ourselves in serious trouble. For example, bananas can not be eaten with cantaloupe, otherwise it will lead to kidney deficiency.

In this project, we are given M types of fruits and their prices presented by ordered numbers and N pairs of fruits indicating which must not be eaten together. Our task is to find the maximum number of safe fruits and print them in a line. If there are multiple solutions , we should output the one with the lowest total price.

The problem is essentially a matter of finding the **maximal clique** in a undirected graph. Thus the **Bron-Kerbosch** algorithm is just the very solution for it. In process, we can make good use of the **back-tracking** concepts we learnt in class in order to optimize the algorithm and reduce the running time.

Chapter 2: Algorithm Specification

2.1 Bron-Kerbosch algorithm

The major algorithm we use is the **Bron-Kerbosch** algorithm. The basic form of the Bron–Kerbosch algorithm is a recursive backtracking algorithm that searches for all maximal cliques in a given graph G. More generally, given three disjoint sets of vertices R, P, and X, it finds the maximal cliques that include all of the vertices in R, some of the vertices in P, and none of the vertices in X. In each call to the algorithm, P and X are disjoint sets whose union consists of those vertices that form cliques when added to R. In other words, $P \cup X$ is the set of vertices which are joined to every element of R. When P and X are both empty there are no further elements that can be added to R, so R is a maximal clique and the algorithm outputs R.

The recursion is initiated by setting R and X to be the empty set and P to be the vertex set of the graph. Within each recursive call, the algorithm considers the vertices in P in turn; if there are no such vertices, it either reports R as a maximal clique (if X is empty), or backtracks. For each vertex v chosen from P, it makes a recursive call in which v is added to R and in which P and X are restricted to the neighbor set $N(v)$ of v, which finds and reports all clique extensions of R that contain v. Then, it moves v from P to X to exclude it from consideration in future cliques and continues with the next vertex in P.

That is, in pseudocode, the algorithm performs the following steps:

Algorithm 1 BronKerbosch1

```
1:  function BronKerbosch1(R, P, X)
2:      if P and X are both empty then
3:          report R as a maximal clique
4:      end if
5:      for each vertex v in P do
6:          BronKerbosch1 (R ∪ {v}, P ∩ N(v), X ∩ N(v))
7:          P := P \ {v}
8:          X := X ∪ {v}
9:      end for
10: end function
```

2.2 Optimizing methods

2.2.1 With pivoting

The basic form of the algorithm, described above, is inefficient in the case of graphs with many non-maximal cliques: it makes a recursive call for every clique, maximal or not. To save time and allow the algorithm to backtrack more quickly in branches of the search that contain no maximal cliques, Bron and Kerbosch introduced a variant of the algorithm involving a "pivot vertex" u, chosen from P (or more generally, as later investigators realized, from $P \cup X$). Any maximal clique must include either u or one of its non-neighbors, for otherwise the clique could be augmented by adding u to it. Therefore, only u and its non-neighbors need to be tested as the choices for the vertex v that is added to R in each recursive call to the algorithm.

In pseudocode:

Algorithm 2 BronKerbosch2

```
1: function BronKerbosch2(R, P, X)
2:     if P and X are both empty then
3:         report R as a maximal clique
4:     end if
5:     choose a pivot vertex u in P ∪ X
6:     for each vertex v in P \ N(u) do
7:         BronKerbosch2 (R ∪ {v}, P ∩ N(v), X ∩ N(v))
8:         P := P \ {v}
9:         X := X ∪ {v}
10:    end for
11: end function
```

2.2.2 With vertex ordering

An alternative method for improving the basic form of the Bron–Kerbosch algorithm involves forgoing pivoting at the outermost level of recursion, and instead choosing the ordering of the recursive calls carefully in order to minimize the sizes of the sets P of candidate vertices within each recursive call.

The degeneracy of a graph G is the smallest number d such that every subgraph of G has a vertex with degree d or less. Every graph has a degeneracy ordering, an ordering of the vertices such that each vertex has d or fewer neighbors that come later in the ordering; a degeneracy ordering may be found in linear time by repeatedly selecting the vertex of minimum degree among the remaining vertices. If the order of the vertices v that the Bron–Kerbosch algorithm loops through is a degeneracy ordering, then the set P of candidate vertices in each call (the neighbors of v that are later in the ordering) will be guaranteed to have size at most d. The set X of excluded vertices will consist of all earlier neighbors of v, and may be much larger than d. In recursive calls to the algorithm below the topmost level of the recursion, the pivoting version can still be used.

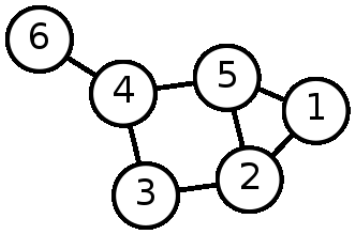
In pseudocode, the algorithm performs the following steps:

Algorithm 3 BronKerbosch3

```
1: function BronKerbosch3(G)
2:     P = V(G)
3:     R = X = empty
4:     for each vertex v in a degeneracy ordering of G do
5:         BronKerbosch2 ({v}, P ∩ N(v), X ∩ N(v))
6:         P := P \ {v}
7:         X := X ∪ {v}
8:     end for
9: end function
```

This variant of the algorithm can be proven to be efficient for graphs of small degeneracy, and experiments show that it also works well in practice for large sparse social networks and other real-world graphs.

There is an example using BronKerbosch algorithm:



In the example graph shown, the algorithm is initially called with $R = \emptyset$, $P = \{1, 2, 3, 4, 5, 6\}$, and $X = \emptyset$. The pivot u should be chosen as one of the degree-three vertices, to minimize the number of recursive calls; for instance, suppose that u is chosen to be vertex 2. Then there are three remaining vertices in $P \setminus N(u)$: vertices 2, 4, and 6.

The iteration of the inner loop of the algorithm for $v = 2$ makes a recursive call to the algorithm with $R = \{2\}$, $P = \{1, 3, 5\}$, and $X = \emptyset$. Within this recursive call, one of 1 or 5 will be chosen as a pivot, and there will be two

second-level recursive calls, one for vertex 3 and the other for whichever vertex was not chosen as pivot. These two calls will eventually report the two cliques $\{1, 2, 5\}$ and $\{2, 3\}$. After returning from these recursive calls, vertex 2 is added to X and removed from P.

The iteration of the inner loop of the algorithm for $v = 4$ makes a recursive call to the algorithm with $R = \{4\}$, $P = \{3, 5, 6\}$, and $X = \emptyset$ (although vertex 2 belongs to the set X in the outer call to the algorithm, it is not a neighbor of v and is excluded from the subset of X passed to the recursive call). This recursive call will end up making three second-level recursive calls to the algorithm that report the three cliques $\{3, 4\}$, $\{4, 5\}$, and $\{4, 6\}$. Then, vertex 4 is added to X and removed from P.

In the third and final iteration of the inner loop of the algorithm, for $v = 6$, there is a recursive call to the algorithm with $R = \{6\}$, $P = \emptyset$, and $X = \{4\}$. Because this recursive call has P empty and X non-empty, it immediately backtracks without reporting any more cliques, as there can be no maximal clique that includes vertex 6 and excludes vertex 4.

The call tree for the algorithm, therefore, looks like:

```
BronKerbosch2(∅, {1,2,3,4,5,6}, ∅)
   BronKerbosch2({2}, {1,3,5}, ∅)
       BronKerbosch2({2,3}, ∅, ∅): output {2, 3}
       BronKerbosch2({2,5}, {1}, ∅)
           BronKerbosch2({1,2,5}, ∅, ∅): output {1,2,5}
   BronKerbosch2({4}, {3,5,6}, ∅)
       BronKerbosch2({3,4}, ∅, ∅): output {3,4}
       BronKerbosch2({4,5}, ∅, ∅): output {4,5}
       BronKerbosch2({4,6}, ∅, ∅): output {4,6}
BronKerbosch2({6}, ∅, {4}): no outpu
```

The graph in the example has degeneracy two; one possible degeneracy ordering is 6,4,3,1,2,5. If the vertex-ordering version of the Bron–Kerbosch algorithm is applied to the vertices, in this order, the call tree looks like:

```
   BronKerbosch3(G)
       BronKerbosch2({6}, {4}, ∅)
           BronKerbosch2({6,4}, ∅, ∅): output {6,4}
       BronKerbosch2({4}, {3,5}, {6})
           BronKerbosch2({4,3}, ∅, ∅): output {4,3}
           BronKerbosch2({4,5}, ∅, ∅): output {4,5}
       BronKerbosch2({3}, {2}, {4})
           BronKerbosch2({3,2}, ∅, ∅): output {3,2}
       BronKerbosch2({1}, {2,5}, ∅)
           BronKerbosch2({1,2}, {5}, ∅)
               BronKerbosch2({1,2,5}, ∅, ∅): output {1,2,5}
   BronKerbosch2({2}, {5}, {1,3}): no output
   BronKerbosch2({5}, ∅, {1,2,4}): no output
```

2.3 Implementation

In this specific problem, we followed the steps shown below to solve the problem.

1. Read in the data and form a undirected graph
2. Initialize the maximum number
3. Use the Bron-Kerbosch algorithm by calling a dfs
4. Undo the process if the search result is invalid
5. Update the answer set if a bigger clique is found
6. Compare the total price of two cliques of the same size and pick the one whcih costs less
7. Output the answer set

```
    int N, M;                       // the N and M
    int G[MAX_N][MAX_N];            // the graph
    int best_price;                 // the least price when the count is most
    int best_cnt;                   // the most count of vertex in maximal clique
    int cur_price;                  // current price
    vector<int> cur_ans;            // current answer set
    vector<int> maxcli_cnt;         // the count of maximal clique from i to M
    vector<int> answer;             // the answer set
    vector<int> id_to_fruit;        // the Transformation from id to fruit
    vector<int> fruit_to_id;        // the Transformation from fruit to id
    vector<int> id_to_price;        // the Transformation from id to price
```

Here we create an adjacency matrix G to present the undirected graph. The $best\_price$ represents the price of the total cost of current answer with the $best\_cnt$ size. The final answer is stored in $answer$. And cur_ans stores the current answer set (of different steps). There are also three vectors indicating the id and price relationships of the fruits.

```
void Bron_kerbosch() {
    /* initial the best_cnt */
    best_cnt = 0;
    /* this loop is used to call the dfs from i = M-1 to i = 0 (decrease) */
    for (int i = M - 1; i >= 0; --i) {
        vector<int> adj;                // the adjacent array of cur vertex
        for (int j = i + 1; j < M; ++j) {
            if (G[i][j] == 0)
                adj.emplace_back(j);    // add the vertex which is adjacent to i
    into this array
        }
        cur_ans.emplace_back(i);                    // "DO"
        cur_price += id_to_price[i];                // "DO"
        Dfs(adj, 1);                                // call the dfs
        cur_price -= id_to_price[i];                // "UNDO"
        cur_ans.pop_back();                         // "UNDO"
        maxcli_cnt[i] = best_cnt;                   // update the maxcli_cnt
    }
}
```

This one of our key functions in this project.

Firstly we initialize the $best\_cnt$ and begin the Bron-Kerbosch loop. We create a new vector adj to store the adjacet array of the current vertex (using the pivoting concepts), add the current certex into the current answer set, update the current cost and call the dfs. At the end of the function, we undo the change we made above.

```
void Dfs(vector<int> &adj, int cur_cnt) {
    /* if adj is empty, means we get a maximal clique */
    if (adj.empty()) {
        /* if this maximal clique is bigger than answer, update the answer
            * if it is as bigger as the answer, compare the price */
        if (cur_cnt > best_cnt || (cur_cnt == best_cnt && cur_price <
   best_price)) {
            answer.assign(cur_ans.begin(), cur_ans.end());
            best_price = cur_price;
            best_cnt = cur_cnt;
        }
        /* other situation has been pruned */
        return;
    }
```

```
    for (int i = 0; i < adj.size(); i++) {
        int cur_vertex = adj[i];

        if (cur_cnt + M - cur_vertex < best_cnt)return;          // prune
        if (cur_cnt + maxcli_cnt[cur_vertex] < best_cnt)return; // prune

        // create a new vector to store the adj of cur vertex
        vector<int> new_adj;
        for (int j = i + 1; j < adj.size(); ++j) {
            // if adj[j] is adj to cur vertex, add it into vector
            if (G[cur_vertex][adj[j]] == 0)
                new_adj.emplace_back(adj[j]);
        }

        cur_ans.emplace_back(cur_vertex);              // "DO"
        cur_price += id_to_price[cur_vertex];          // "DO"
        Dfs(new_adj, cur_cnt + 1);                     // call dfs recursive
        cur_price -= id_to_price[cur_vertex];          // "UNDO"
        cur_ans.pop_back();                            // "UNDO"
    }
}
```

The dfs function is the further implementation of the Bron-Kerbosch algorithm. If the adj array is empty, it means that the search ends and the cur_ans stores the current answer set. We just need to compare the size of the current answer set and the best answer set. If the current answer set is bigger, update the parameters of the best answer set. If they are of the same size, compare their total cost and pick the lower one. Otherwise we just discard the current answer set.

The second part of the dfs implement the pivoting prunning and degeneracy prunning of the Bron-Kerbosch algorithm. We can tactfully use the index of the vertex to judge if further search will geneate a better answer. The rest is the same as the $Bron\_Kerbosch$ function.

Chapter 3: Testing Results

3.1 Simple testing cases

3.1.1 Case1: Minimum case

This case only contains one vertex and no tips, to see if the program output the vertex.
INPUT

```
0 1
001 3
```

OUTPUT

```
1
001
3
```

3.1.2 Case2: All selected

This case contains no tips, to see if the program select all the vertex.
INPUT

```
0 5
005 1
004 1
```

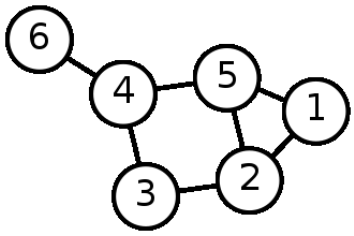
```
003 1
002 1
001 2
```

OUTPUT

```
5
001 002 003 004 005
6
```

3.1.3 Case3: Normal simple case - without comapring total price

This is a normal case of the following graph. This case has only one maximal clique and there's no need to compare the fruits' prices.



INPUT

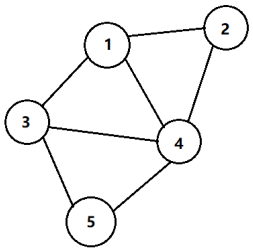
```
8 6
001 006
002 006
003 006
005 006
001 004
002 004
003 005
001 003
006 1
005 2
004 1
003 5
002 2
001 2
```

OUTPUT

```
3
001 002 005
6
```

3.1.4 Case4: Normal simple case - with comapring total price

This is a normal case of the following graph. This case has multiple maximal cliques and we need to compare the total prices of each clique.



INPUT

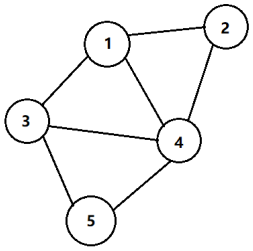
```
3 5
001 005
002 003
002 005
005 5
004 4
003 3
002 2
001 1
```

OUTPUT

```
3
001 002 004
7
```

3.1.5 Case5: With invalid tips

In this case, there some invalid tips which means that some indexes in the tips did not appear in the price list, which should be considerated.



INPUT

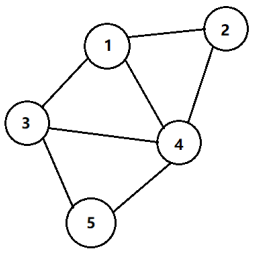
```
4 5
001 005
002 003
002 005
007 008
005 5
004 4
003 3
002 2
001 1
```

OUTPUT

```
3
001 002 004
7
```

3.1.6 Case6: All invalid tips - select all

The case's tips are all invalid, so the program also need to pick all the fruits.



INPUT

```
4 5
010 011
012 050
001 030
002 009
005 5
004 4
003 3
002 2
001 1
```

OUTPUT

```
5
001 002 003 004 005
15
```

3.2 Large data

Use order "$python\ testcase\_generator.py$" in the terminal and enter 2 numbers N and M (no more than 100). This will create a $input.txt$ with random input. Then you can use this input to test our program.

There is an example of large data test in the hand-in zip named $test\_case1.txt$ and $output1.txt$ (with $N = 100\ M = 50$).

Here is the **Run time table** of some data we tested:

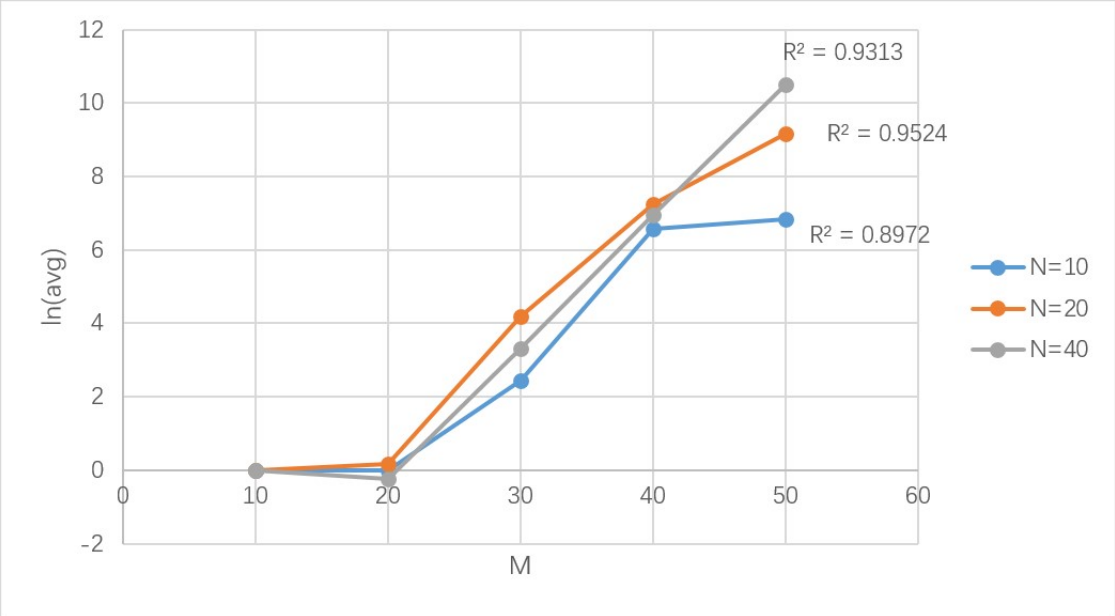


Figure 1: Run time table

M	10	20	30	40	50
ticks1	0	2	10	755	147
ticks2	0	1	13	294	1363
ticks3	0	1	10	2122	376
ticks4	0	0	2	27	893
ticks5	0	1	22	386	1832
avg	0	1	11.4	716.8	922.2
ln(avg)	n/a	0	2.4336	6.5748	6.8268

N=10

M	10	20	30	40	50
ticks1	0	2	89	750	13401
ticks2	0	1	22	593	1481
ticks3	0	1	12	4016	526
ticks4	0	1	74	660	27865
ticks5	0	1	130	984	4172
avg	0	1.2	65.4	1400.6	9489
ln(avg)	n/a	0.1823	4.1805	7.2447	9.1579

N = 20

M	10	20	30	40	50
ticks1	0	1	25	411	5997
ticks2	0	1	21	1395	15597
ticks3	0	1	36	478	75457
ticks4	0	0	28	2434	20683
ticks5	0	1	26	442	66126
avg	0	0.8	27.2	1032	36772
ln(avg)	n/a	-0.2231	3.3032	6.9393	10.5125

N=40

3.3 PTA submission

Submit Time	Status	Score	Problem	Compiler	Run Time
[illegible]	Accepted	35	1021	[illegible]	[illegible]

Case	Result	Run Time	Memory
0	Accepted	[illegible]	[illegible]
1	Accepted	[illegible]	[illegible]
2	Accepted	[illegible]	[illegible]
3	Accepted	[illegible]	[illegible]
4	Accepted	[illegible]	[illegible]
5	Accepted	[illegible]	[illegible]
6	Accepted	[illegible]	[illegible]

Chapter 4: Analysis and Comments

4.1 Correctness proof

Three sets play an important role in the algorithm. (1) The set compsub is the set to be extended by a new point or shrunk by one point on traveling along a branch of the backtracking tree. The points that are eligible to extend compsub, i.e. that are connected to all points in compsub, are collected recursively in the remaining two sets. (2) The set candidates is the set of all points that will in due time serve as an extension to the present configuration of compsub. (3) The set not is the set of all points that have at an earlier stage already served as an extension of the present configuration of compsub and are now explicitly excluded. The reason for maintaining this set trot will soon be made clear.

The core of the algorithm consists of a recursively defined extension operator that will be applied to the three sets Just described. It has the duty to generate all extensions of the given configuration of compsub that it can make with the given set of candidates and that do not contain any of the points in not. To put it differently: all extensions of compsub containing any point in not have already been generated. The basic mechanism now consists of the following five steps:

- Step 1. Selection of a candidate.
- Step 2. Adding the selected candidate to compsub.

- Step 3. Creating new sets candidates and not from the old sets by removing all points not connected to the selected candidate (to remain consistent with the definition), keeping the old sets in tact.
- Step 4. Calling the extension operator to operate on the sets just formed.
- Step 5. Upon return, removal of the selected candidate from compsub and its addition to the old set not.

We will now motivate the extra labor involved in maintaining the sets not. A necessary condition for having created a clique is that the set candidates be empty; otherwise compsub could still be extended. This condition, however, is not sufficient, because if now not is nonempty, we know from the definition of not that the present configuration of compsub has already been contained in another configuration and is therefore not maximal. We may now state that compsub is a clique as soon as both not and candidates are empty.

If at some stage trot contains a point connected to all points in candidates, we can predict that further extensions (further selection of candidates) will never lead to the removal (in Step 3) of that particular point from subsequent configurations of not and, therefore, not to a clique. This is the branch and bound method which enables us to detect in an early stage branches of the backtracking tree that do not lead to successful endpoints.

A few more remarks about the implementation of the algorithm seem in place. The set compsub behaves like a stack and can be maintained and updated in the form of a global array. The sets candidates and not are handed to the extensions operator as a parameter. The operator then declares a local array, in which the new sets are built up, that will be handed to the inner call. Both sets are stored in a single one-dimensional array with the following layout:

$$\Big|\ not \qquad \Big|\ candidates$$

index values:$I \ldots ne \ldots ce$

The following properties obviously hold:

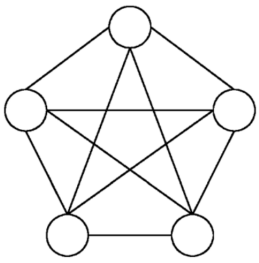
1. $ne \leq ce$
2. $ne = ce$:empty ($candidates$)
3. $ne = 0$:empty(not)
4. $ce = 0$:empty(not) and empty ($candidates$) = clique found

If the selected candidate is in array position ne q- 1, then the second part of Step 5 is implemented as ne : = ne + 1.

4.2 Complexity Analysis

4.2.1 Space Complexity

In this project we use the DFS algorithm to implement the Bron–Kerbosch algorithm, which needs a recursive work stack. The worst case is that under a complete graph, the search process must go through all the vertex on the graph, so its space complexity is $O(M)$.



Other than that, the undirected graph of the relationship between the fruits is account for the main space cost. The graph is built using M nodes and is of size $M \times M$, thus its space complexity is $O(M^2)$. The rest of the parameters cost no more than $O(M)$. Thus the space complexity is $O(M^2)$.

4.2.2 Time Complexity

The Bron–Kerbosch algorithm is not an output-sensitive algorithm: unlike some other algorithms for the clique problem, it does not run in polynomial time per maximal clique generated. However, it is efficient in a worst-case sense, any n-vertex graph has at most $\frac{3^n}{3}$ maximal cliques, and the worst-case running time of the Bron–Kerbosch algorithm (with a pivot strategy that minimizes the number of recursive calls made at each step) is $O(\frac{3^n}{3})$, matching this bound.

For sparse graphs, tighter bounds are possible. In particular the vertex-ordering version of the Bron–Kerbosch algorithm can be made to run in time $O(dn3^{\frac{d}{3}})$, where d is the degeneracy of the graph, a measure of its sparseness. There exist d-degenerate graphs for which the total number of maximal cliques is $(n-d)3^{\frac{d}{3}}$, so this bound is close to tight.

In the **Run time table**, we have already done a linear fitting for the ticks we got. We can see that three R^2 are very close to 1 with least being 0.8972, which also fits this conclusion.

Appendix: Source Code (in C++)

```
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <vector>
#include <string>

/* This is a switch for switching the input method */
//#define _TEST_

#define MAX_N 1024

using namespace std;

int N, M;                          // the N and M
int G[MAX_N][MAX_N];               // the graph
int best_price;                    // the least price when the count is most
int best_cnt;                      // the most count of vertex in maximal clique
int cur_price;                     // current price
vector<int> cur_ans;               // current answer set
vector<int> maxcli_cnt;            // the count of maximal clique from i to M
vector<int> answer;                // the answer set
vector<int> id_to_fruit;           // the Transformation from id to fruit
vector<int> fruit_to_id;           // the Transformation from fruit to id
vector<int> id_to_price;           // the Transformation from id to price

void Bron_kerbosch();              // the function of Bron Kerbosch Algorithm

void Dfs(vector<int> &adj, int cur_cnt);// the function of dfs which is a part
    of Bron Kerbosch

void Print_ans();                  // Print the final answer

int main() {
#ifdef _TEST_                                  // file stream input and output
    freopen("./test_case.txt", "r", stdin);// You can store the input data in
    this file
    freopen("./output.txt", "w", stdout);  // the output data will be output
    into this file
#endif
    int fruit1, fruit2;                        // temp variable
    int fruit, price;                          // temp variable
    vector<pair<int, int> > tips;              // temp variable used to store tips
```

```
    /* initial the global variables */
    id_to_fruit.resize(MAX_N);
    fruit_to_id.resize(MAX_N);
    id_to_price.resize(MAX_N);
    maxcli_cnt.resize(MAX_N);

    cin >> N >> M;                          // read in the N and M
    for (int i = 0; i < 1024; ++i) {        // initial the fruit_to_id array
        fruit_to_id[i] = -1;                // -1 means this index is not used
    }

    for (int i = 0; i < N; ++i) {
        cin >> fruit1 >> fruit2;            // read the tips
        tips.emplace_back(make_pair(fruit1, fruit2));
    }

    for (int i = 0; i < M; ++i) {
        cin >> fruit >> price;              // read the fruit and price
        fruit_to_id[fruit] = i;             // store the mapping relations
        id_to_price[i] = price;             // store the mapping relations
        id_to_fruit[i] = fruit;             // store the mapping relations
    }

    for (int i = 0; i < N; ++i) {
        G[i][i] = 1;
        /* if A and B is adjacent then G[A][B] = G[B][A] = 1  */
        if (fruit_to_id[tips[i].first] != -1 && fruit_to_id[tips[i].second] !=
   -1) {
            G[fruit_to_id[tips[i].first]][fruit_to_id[tips[i].second]] = 1;
            G[fruit_to_id[tips[i].second]][fruit_to_id[tips[i].first]] = 1;
        }
    }

    Bron_kerbosch();
    Print_ans();
}

void Bron_kerbosch() {
    /* initial the best_cnt */
    best_cnt = 0;
    /* this loop is used to call the dfs from i = M-1 to i = 0 (decrease) */
    for (int i = M - 1; i >= 0; --i) {
        vector<int> adj;                // the adjacent array of cur vertex
        for (int j = i + 1; j < M; ++j) {
            if (G[i][j] == 0)
                adj.emplace_back(j);    // add the vertex which is adjacent to i
    into this array
        }
        cur_ans.emplace_back(i);                    // "DO"
        cur_price += id_to_price[i];                // "DO"
        Dfs(adj, 1);                                // call the dfs
        cur_price -= id_to_price[i];                // "UNDO"
        cur_ans.pop_back();                         // "UNDO"
        maxcli_cnt[i] = best_cnt;                   // update the maxcli_cnt
    }
}
```

```
void Dfs(vector<int> &adj, int cur_cnt) {
    /* if adj is empty, means we get a maximal clique */
    if (adj.empty()) {
        /* if this maximal clique is bigger than answer, update the answer
         * if it is as bigger as the answer, compare the price */
        if (cur_cnt > best_cnt || (cur_cnt == best_cnt && cur_price <
   best_price)) {
            answer.assign(cur_ans.begin(), cur_ans.end());
            best_price = cur_price;
            best_cnt = cur_cnt;
        }
        /* other situation has been pruned */
        return;
    }

    for (int i = 0; i < adj.size(); i++) {
        int cur_vertex = adj[i];

        if (cur_cnt + M - cur_vertex < best_cnt)return;          // prune
        if (cur_cnt + maxcli_cnt[cur_vertex] < best_cnt)return; // prune

        // create a new vector to store the adj of cur vertex
        vector<int> new_adj;
        for (int j = i + 1; j < adj.size(); ++j) {
            // if adj[j] is adj to cur vertex, add it into vector
            if (G[cur_vertex][adj[j]] == 0)
                new_adj.emplace_back(adj[j]);
        }

        cur_ans.emplace_back(cur_vertex);              // "DO"
        cur_price += id_to_price[cur_vertex];          // "DO"
        Dfs(new_adj, cur_cnt + 1);                     // call dfs recursive
        cur_price -= id_to_price[cur_vertex];          // "UNDO"
        cur_ans.pop_back();                            // "UNDO"
    }
}


void Print_ans() {
    vector<int> answer_out;                // create a vector to store the
   fruit of answer
    for (auto &iter:answer)                // convert id to fruit
        answer_out.emplace_back(id_to_fruit[iter]);
    sort(answer_out.begin(), answer_out.end());       // sort the answer set
    cout << best_cnt << endl;              // print the number of answer set
    for (auto &iter:answer_out)            // print item of the answer set
        cout << setw(3) << setfill('0')
             << iter << ((iter == answer_out.back()) ? "" : "␣");
    cout << endl << best_price;            // print the total price of answer
   set
}
```

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:

Omitted