Project 3 Safe Fruit

- Lin Juyi - 3180103721
- Lu Tian - 3180103740
- Gu Yu - 3190105872

1. Description of the problem.

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 (哈密瓜), otherwise it will lead to kidney deficiency (肾虚).

Now you are given a long list of such tips, and a big basket of fruits. You are supposed to pick up those fruits so that it is safe to eat any of them.

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 idToIndex more than 100.

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 idToIndex 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 idToIndex more than 1000. All the numbers in a line are separated by spaces.

Output Specification:

For each case, first print in a line the maximum number of safe fruits. Then in the next line list all the safe fruits, in increasing order of their ID's. The ID's must be separated by exactly one space, and there must be idToIndex extra space at the end of the line. Finally in the third line print the total price of the above fruits. Since there may be many different solutions, you are supposed to output the one with a maximum number of safe fruits. In case there is a tie, output the one with the lowest total price. It is guaranteed that such a solution is unique.

In our input file, we do not guarantee that such a solution is unique.

2. Algorithm

The maximum independent set is the set of all fruits that are not connected to each other. We require the maximum independent set to be the set of vertices V with K vertices that are not connected to each other.

The complement of a graph G, in layman's terms, is the graph Kn-G obtained by removing the set of edges of G from the complete graph Kn. It is a graph with the same points as *G*, and these points are connected by edges when and only when *G* they are not connected by edges.

Maximal cluster: The set V of vertices takes K vertices which are connected by edges between two of them.

The number of vertices in the maximal cluster = the number of vertices in the maximal independent set of the complementary graph

By finding the maximum number of vertices in the maximal cluster of the complementary graph, we can obtain the maximum number of vertices in the original graph.

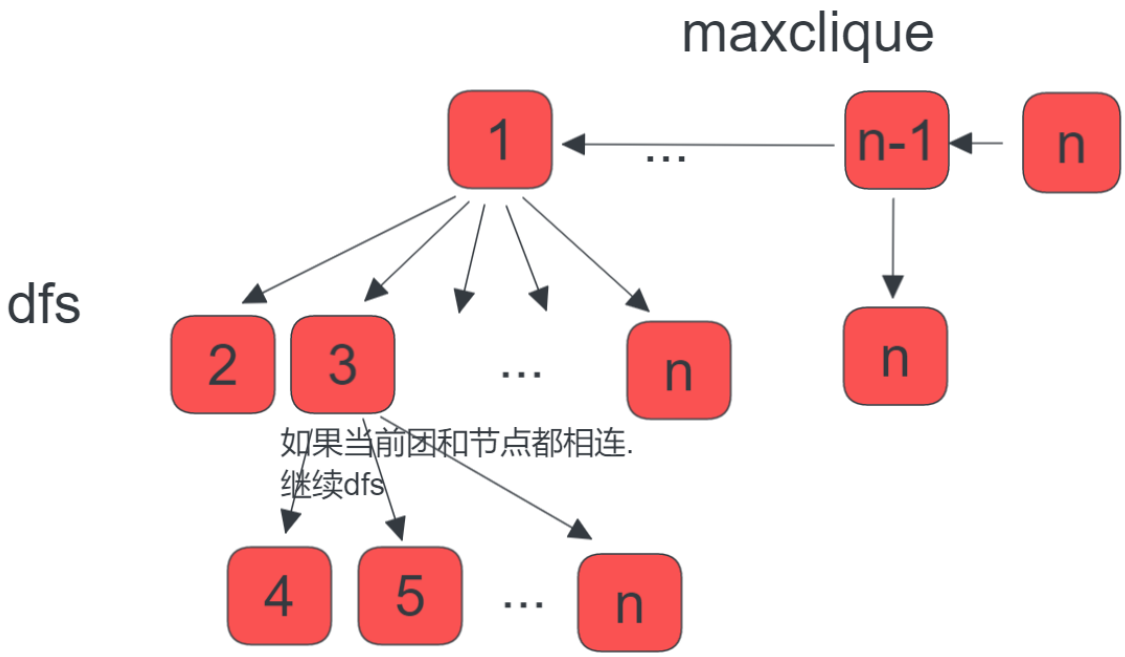
First, set all the adjacency matrices to 1, and then set the input edges to 0, so that the complementary graph is obtained.

How to branch and how to prune:

Iterate through all vertices from back to front, vis as an array of paths, also called path.

if all nodes in current cluster connected with current node, then we continue dfs.

Else, we prune this path.



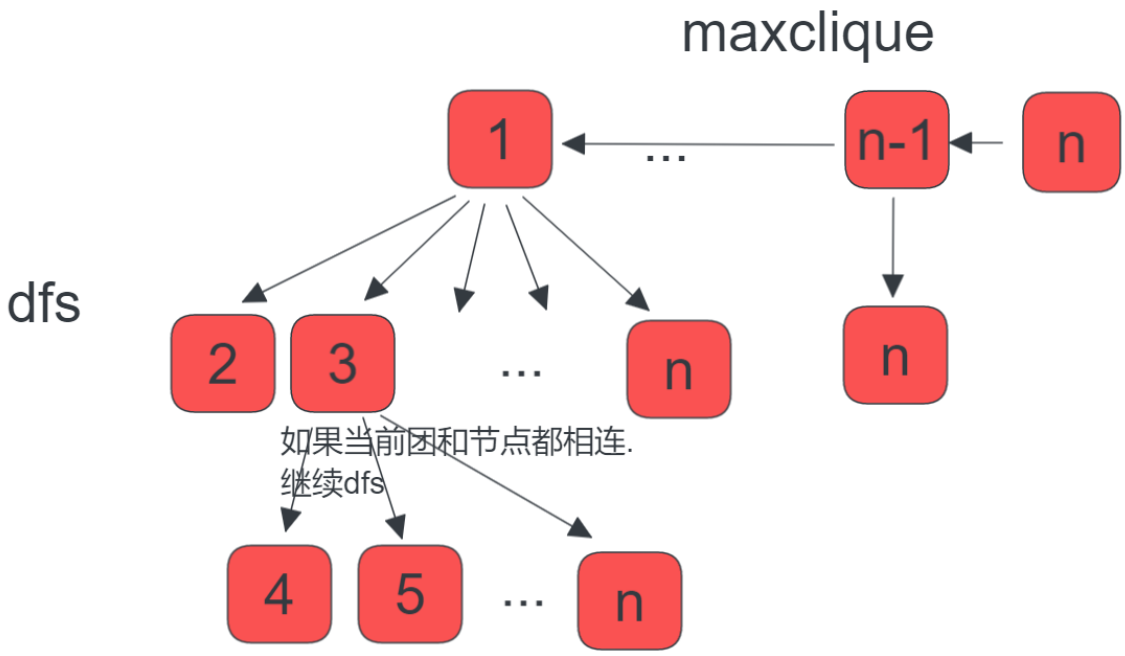
pseudo-code

```
dfs(int current node, int current maximum cluster size,):
    for i from the current node u to the end:
        if latter max + current pos < existing answer:
            return
        if u and i are connected:
            Query if every node in the cluster is connected to i
        if all are connected:
            Add i to the current max group.
            dfs recursively down
    if this time number > ans:
        Record in the group array
        ans = number this time
        return true
    return false
solve (int n ):
    ans = -1
    for i in range(back,front):
        Start dfs with the largest cluster of size 1
        The ans of this node is recorded in cnt.
    If ans is large, return
```

3. Theoretical Analysis

Bounds of the running time of our algorithm on a graph with node can be expressed as a function of node

From figure 1 we can find:



The tree root 1:

f(n) = f(1)+ f(2) +... + f(n-1) => f(n) = 2 f(n-1)

The size of tree n :

2^n

The size of total algorithm:

2^(n-1) + 2^(n-2) + .. + 2 +1 =2^n

Thus, running time is O(2^n)

In the case of very sparse edges, most of the branches were cut. Thus, our algorithm don't perform well in graph which has more edges.

4. Experiment

testing data

When the test data is randomly generated, it is stored in structure array first, and then the structure array is arranged from small to large by bubble sort, and then written into the file and saved

1. Enter relation number m and fruit number n
2. Read m group relation, if the newly generated relation already exists (here using a while loop, when the new relation is not repeated break, otherwise infinite loop), then regenerate and save into the array
3. Sort according to the number size of the first fruit in the relational structure
4. Read the rearranged structure array into the created file
5. Read n random numbers not more than 1000 and save them in the file in descending order

6. Perform data tests with the data in the file

We also generate sparse graph, #edges = sqrt(# vertex)

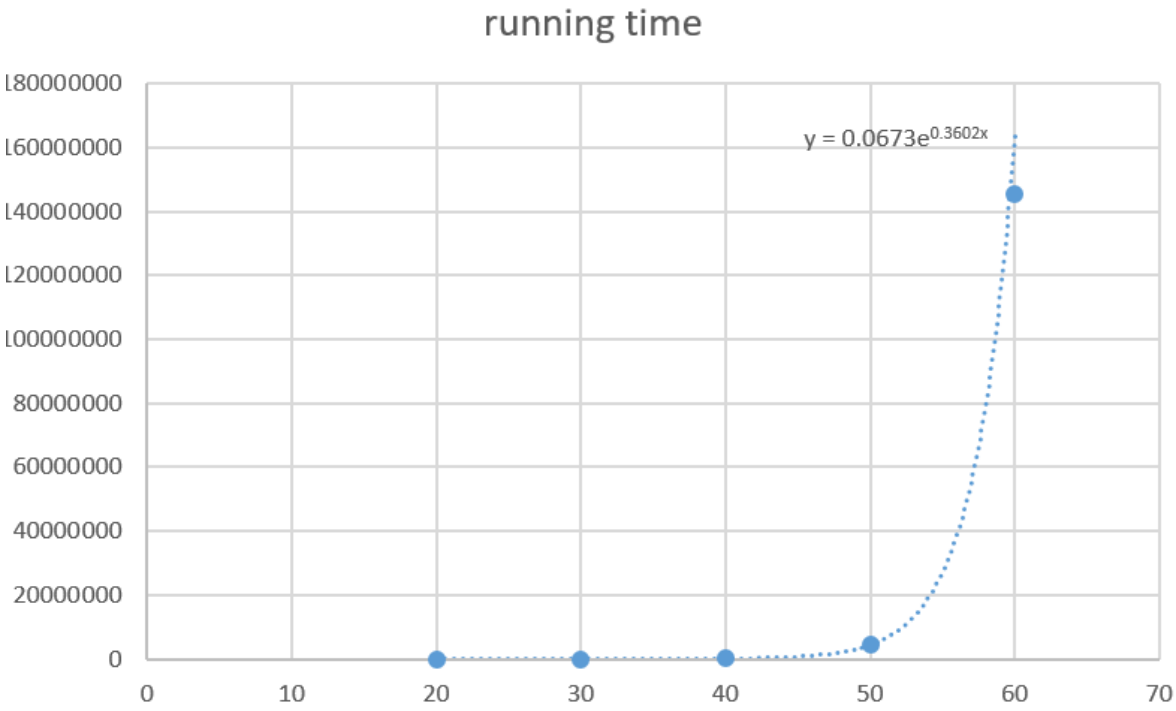
The setting of experiment:

```
auto start = system_clock::now();
    maxclique();
    auto end   = system_clock::now();
    auto duration = duration_cast<microseconds>(end - start);
then we record
    double(duration.count()) *microseconds::period::num
  microseconds::period::num = 10^6
```

node num	running time
20	72.7
30	3947.778
40	147418
50	4313020
60	1.46E+08

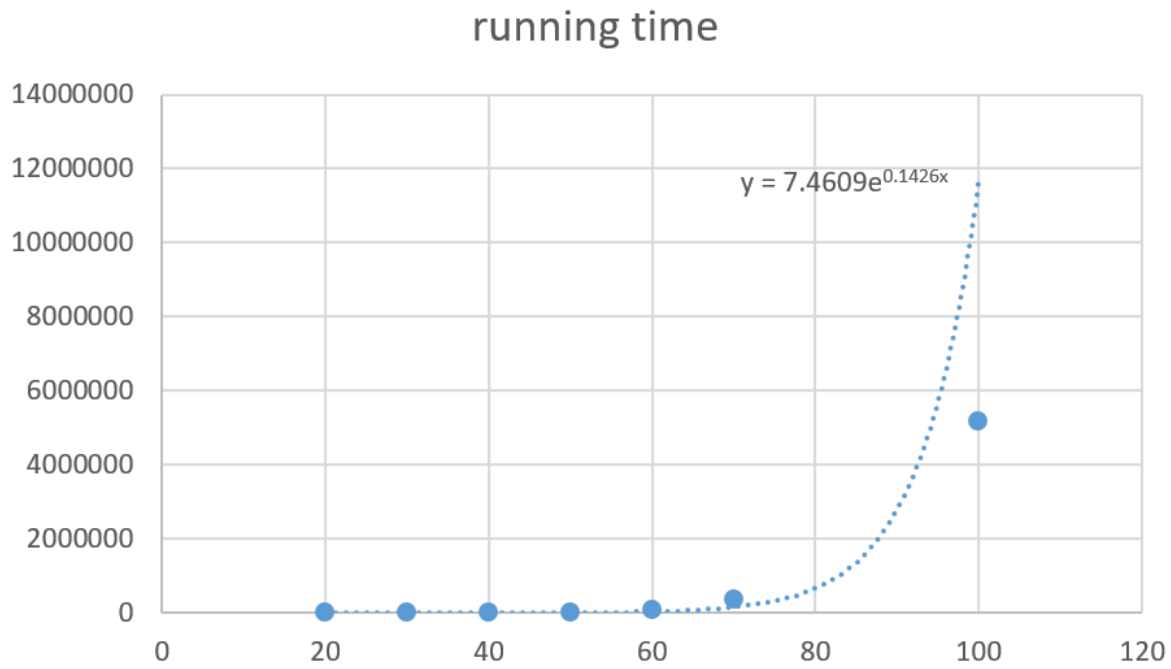
We can find that the graphs which vertex number equals edges number running time is growing exponentially.

Twenty nodes were run 10 times and averaged, 30 and 40 nodes were run 9 times and averaged, 50 nodes were run 4 times and averaged, and 60 nodes were run once.

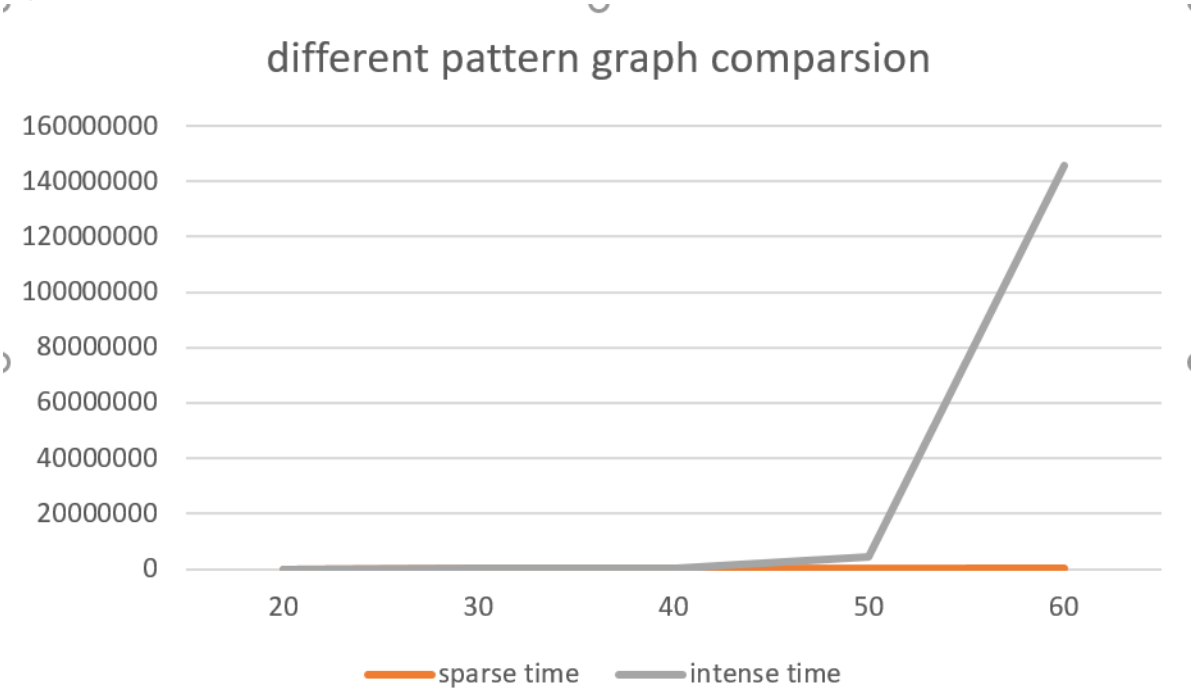


We also implement our experiment in the sparse graphs.

node num	running time
20	79.42857
30	802.8333
40	724.75
50	15921.5
60	79528.5
70	349416
100	5.18E+06



We can find running time in the 100 nodes graph is smaller than the exponential line, our prune algorithm has an obvious effect.

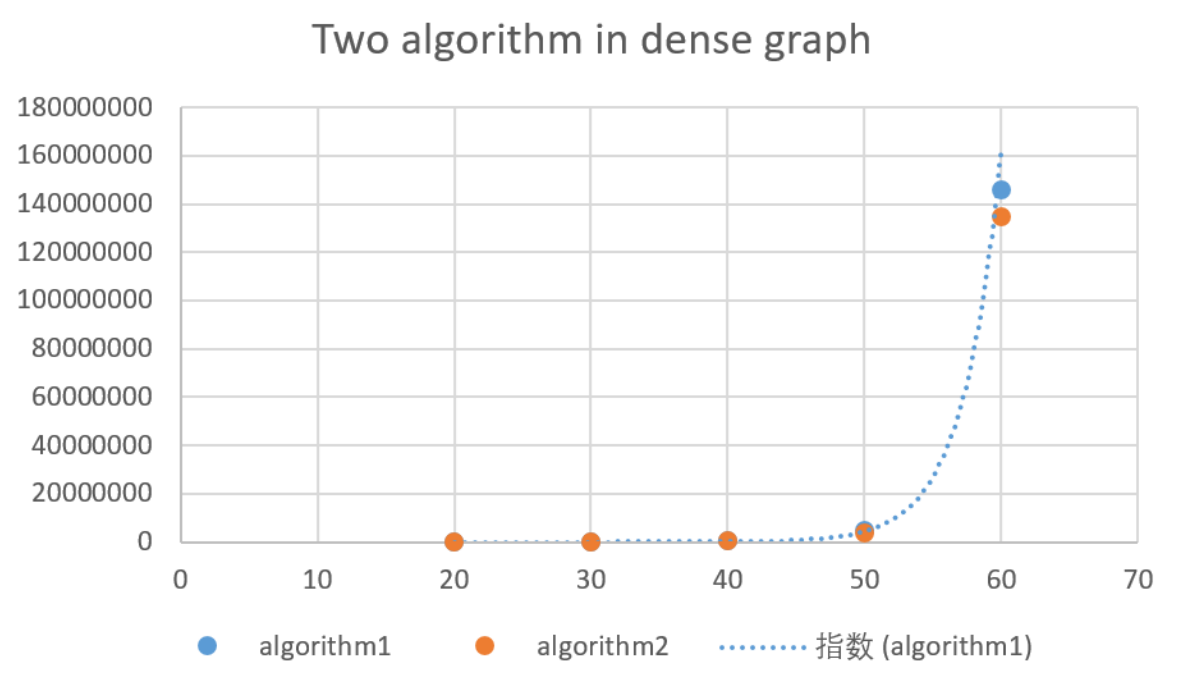


We can find that our algorithm has much better performance in sparse time. It proves that prune algorithm is useful.

Bonus:

We compare with another algorithm. It use a different prune strategy.

```
        if (step + vertexNum - k < best)//new prune strategy
            return;
        if (step + dp[k] < best)
            return;
       // don't have maxcost prune in first algorithm
```



In dense graph, the running time is following:

node num	algorithm1	algorithm2
20	72.7	84
30	3947.78	4700
40	147418	129079
50	4313020	3.68E+06
60	1.5E+08	1.34E+08

In 40, 50 and 60 vertexes graph, new algorithm is quicker than the first algorithm. In 20,30 vertexes graph, new algorithm is slower than the first algorithm. It is because new algorithm has bigger constant items.

In sparse graph, the running time is following:

node num	algorithm1	algorithm2
20	79.4286	110
30	802	950
40	724	828
50	15921	23683
60	79528	53904
70	349416	836378
100	5180765	5.36E+06

In all input sizes, the new algorithm is slower than the first algorithm.

The new prune strategy doesn't have an obvious effect on Sparse graphs because it doesn't have many edges. But in dense graphs, it works well.

5. Discussion and Conclusion

In the case of very sparse edges, most of the branches were cut. Thus, our algorithm doesn't perform well in intense graphs.

The exponential level of growth is very significant, especially when there are many edges.

We can find that our algorithm has much better performance in sparse time. It proves that prune algorithm is useful.

We should test the data more often, and also compare it in more cases.

6. References

1.http://i.stanford.edu/pub/cstr/reports/cs/tr/76/550/CS-TR-76-550.pdf