Backtracking Algorithm: To Buy or Not to Buy

Group number: Jiefeng Wu Jiajun Qin Wenjie Huang

April 15, 2023

Abstract

This report aims to solve a problem involving backtracking algorithms related to string matching. The importance of the problem is highlighted,as the backtracking algorithm is an important tool in computer science and mathematical problem-solving, which is especially useful for solving complex problems that involve a large number of possibilities or constraints, such as the traveling salesman problem, Sudoku puzzles, or the N-queens problem. The methods used in the report include designing a backtracking algorithm and consider pruning as much as possible, testing time spent using the control variable method and analyzing the time complexity and spatial complexity of the algorithms used.In the theoretical analysis, the time complexity of the algorithm is O(2^N * M), where N is the number of strings and M is the maximum length of the strings. The space complexity is O(N*T), where N is the number of strings, and T is the number of colors. However, in practice, we found that when M is large, as N increases, so does contingency, which makes a small number of test samples unable to accurately reflect the relationship between time complexity and M.The implications and significance of this project are discussed, including a better understanding of backtracking algorithms and helping develop general problem-solving skills and intuition for optimization.

1 Introduction

This report aims to solve the problem of finding the minimum number of extra beads Eva needs to buy to make a string of beads with her favorite colors when the shop only sells whole strings. The problem involves selecting the optimal set of bead strings to minimize extra bead purchases, which can be solved by backtracking algorithm.

2 Data Structure / Algorithm Specification

2.1 The Backtracking Algorithm

Backtracking algorithm is a problem-solving technique that involves exploring all possible solutions to a problem by incrementally building a solution and backtracking when the current solution is no longer feasible or complete.

The key idea is to try out various options one by one, and if a particular option does not lead to a valid solution, then backtrack to the previous step and try another option. This process is repeated until a solution is found or all possible options have been explored.

Pseudocode of Backtracking Algorithm used in this project

```
procedure DFS(k, extra, left)
if extra ≥ min_extra then /* Pruning 1 */
return
end if
if left = 0 then     /* Pruning 2 */
```

```
    flag ← true
    min_extra ← extra
    return
end if

if k = N then   /* Boundary */
    if min_left > left then   /* Update min_left */
        min_left ← left
    end if
    return
end if

save_extra ← extra
save_left ← left

/* To buy */
bj ← false
delta[MAXT] ← {0}    /* Record the changed amounts of the beads */
for each i in a[k] do
    if target_beads[i] then
        if Hash[k][i] $\geq$ target_beads[i] then   /* More beads than Eva needs */
            delta[i] ← target_beads[i]
            left ← left - delta[i]
            extra ← extra + Hash[k][i] - target_beads[i]
        else    /* Less beads than Eva needs */
            delta[i] ← Hash[k][i]
            left ← left - delta[i]
        end if
        bj ← true   /* Indicate whether this string can cut down the amount that we
        still need to buy */
    else
        extra ← extra + Hash[k][i]
    end if
end for

/* Pruning 3 */
if bj then
    /* Suppose we choose this string, then we can modify the beads that we still need
    to buy */
    for i ← 0 to MAXT-1 do
        target_beads[i] ← target_beads[i] - delta[i]
    end for

    DFS(k+1, extra, left)

    /* Backtracking, restoring to the state before we modify */
    for i ← 0 to MAXT-1 do
        target_beads[i] ← target_beads[i] + delta[i]
    end for
end if

/* Pruning 4 */
DFS(k+1, save_extra, save_left)   /* Not to buy */
return
```

```
end procedure
```

2.2 The Data structure——array

In this project, we used a one-dimensional array (target-beads[62]) to store the number of bead colors Eva wanted, a two-dimensional array (Hash[$MAXN$][62]) to store the number of various colors in each string that belong to the shop, and an integer two-dimensional container (a[$MAXN$]) to help store the labels of all bead colors in each string, where the $MAXN$ means the maximum number of the strings in the shop. It is worth noting that for convenience, we convert the characters used to represent the color of the beads into consecutive integers (0-9:0-9; a-z:10-35; A-Z:36-61) so that we can use it as the index of array.
Obviously we can use the container map, but it will spend much time so we give up it.

2.3 The Pruning

As we all know, pruning has great significance for backtracking algorithms, which can reduce the search space of backtracking algorithms by eliminating branches that cannot lead to a valid solution. This makes the algorithm more efficient, especially when dealing with large or complex problems.

In this project, we used pruning algorithms four times in total. Pruning 1: When the current "extra" has been bigger than the existing answer, then there is no need to search this branch further. Pruning 2: When you find all the colors Eva wants, just update the extra at this point and return, there is also no need to search this branch further. Pruning 3: When none of the colors of a certain bead is what Eva wants, there is no need to continue searching down (based on buying the bead).Pruning 4: Based on the searching order: first we suppose to buy this string, then we search for the situation that we don't buy it. The advantage of the modified order is that if the answer is in the shallow layer of searching tree, we needn't spend much time on useless branches.

3 Testing Results

3.1 General Test

We guessed that the time complexity of the algorithm used in this project is related to the number of strings(N) and the maximum length of each string(M), so we did the following tests with control variables, as shown in the following two tables (the results are the average time after multiple trials).The sample text file for the test is placed in the folder.

The first test result in Table 2:

Table 1: M=100

N	CLOCK	TIME
2	1	0.001
3	1	0.001
4	2	0.002
6	1	0.001
8	1	0.001
12	1	0.001
16	4	0.004
24	24	0.024
32	148	0.148
48	26209	26.209
64	60420	60.420

Noting that when N is small, the time spent is small and the relationship with N is not significant, we wanted to better observe the relationship between the two by trying to increase M. So we increased M to 1000 in the hope of getting better results.

The second test result in Table 2:

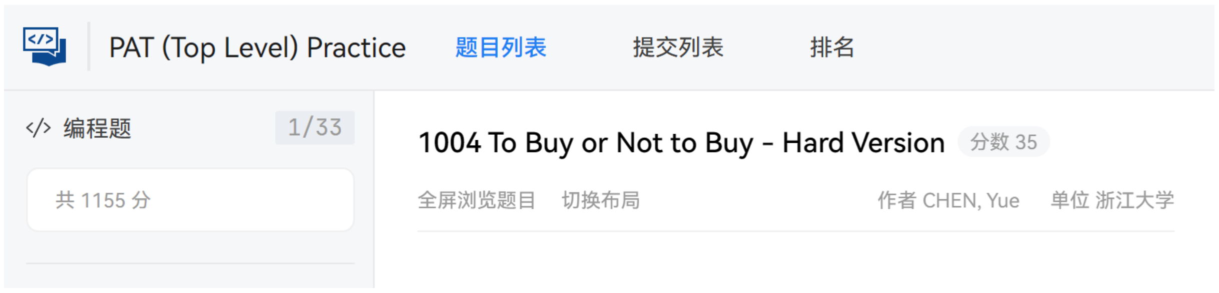
Table 2: M=1000

N	CLOCK	TIME
2	1	0.001
3	2	0.002
4	2	0.002
6	3	0.003
8	3	0.003
16	3	0.003
24	16	0.016
32	55	0.055
48	490	0.490
64	5587	5.587

Summary: However, in the process of testing, we found that when M becomes larger, the length of the target string (what Eva wants) and N test strings increase at the same time, and the increase of the test string length is significantly greater than the increase of the target string length. Therefore, when performing depth-first search, the probability of being pruned and ending the program early is greatly increased, so that when N gradually increases, the average time spent not only does not increase, but shows a decreasing trend. And the randomness is greater, that is, when both N and M are the same, the deviation of the time spent by different test samples will show a greater chance factor.

3.2 PTA Test

In addition to the regular tests, we also test the correctness of the program on the PTA, the results of which are shown in the figure below.



PAT (Top Level) Practice 题目列表 提交列表 排名

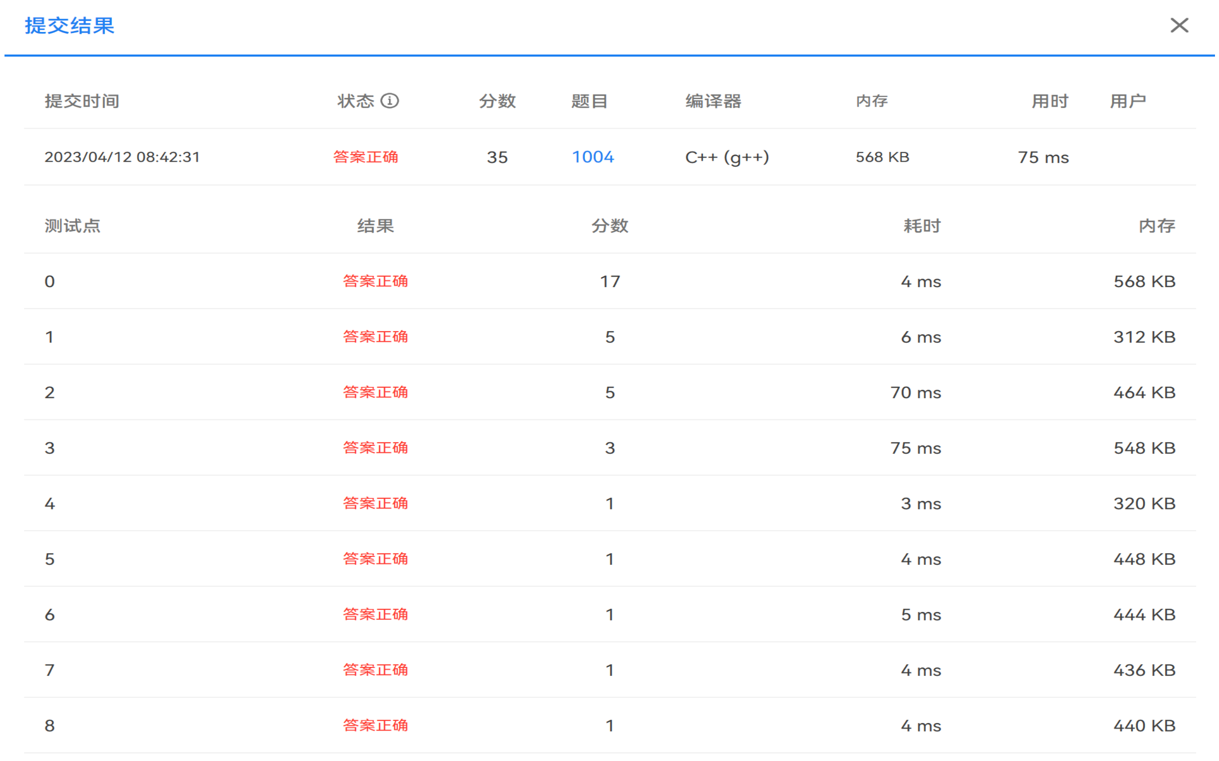
编程题 1/33

共 1155 分

1004 To Buy or Not to Buy - Hard Version 分数 35

全屏浏览题目 切换布局 作者 CHEN, Yue 单位 浙江大学

Figure 1: Source of the problem



提交结果

提交时间	状态	分数	题目	编译器	内存	用时	用户
2023/04/12 08:42:31	答案正确	35	1004	C++ (g++)	568 KB	75 ms	

测试点	结果	分数	耗时	内存
0	答案正确	17	4 ms	568 KB
1	答案正确	5	6 ms	312 KB
2	答案正确	5	70 ms	464 KB
3	答案正确	3	75 ms	548 KB
4	答案正确	1	3 ms	320 KB
5	答案正确	1	4 ms	448 KB
6	答案正确	1	5 ms	444 KB
7	答案正确	1	4 ms	436 KB
8	答案正确	1	4 ms	440 KB

Figure 2: Test results

4 Analysis and Comments

This program is designed to solve a string matching problem. Based on the implementation of the DFS function in the program, it can be seen that this is an algorithm based on depth-first search. It enumerates all possible combinations to find strings that meet the requirements. However, since DFS is an exhaustive algorithm, its time and space complexity may be high, depending on the problem size and the complexity of the algorithm implementation.

4.1 Time Complexity

The time complexity of the provided program is exponential, which means it grows very quickly with the size of the input. Specifically, the time complexity is $\mathbf{O}(2^N * \mathbf{M})$, where N is the number of strings and M is the maximum length of the strings. This is because the program uses a depth-first search algorithm to check all possible combinations of strings that can be used to make up the target string. Since there are 2^N possible combinations, and each combination involves iterating through all the characters in each string (which takes M time), the overall time complexity is $\mathbf{O}(2^N * \mathbf{M})$.
Tips:
In practice, however, the increase in M increases the contingency of test samples. So when the number of tests is small, the time spent shown by the test results has little correlation with M.

4.2 Space Complexity

In the program, the main source of space complexity is the storage of information about each string, including their length, color, and quantity. In addition, the program uses recursion to implement DFS, so the depth of recursion is also an important factor in space complexity. Furthermore, the program also uses a two-dimensional array Hash, which is used to record the number of beads of each color in each string, so its space complexity is **O(N*T)**, where N is the number of strings, and T is the number of colors.

5 Author list

The code and report are finished by all of us.

Jiajun Qin finish the code.

Wenjie Huang and JieFeng Wu finish the report.

Declaration

We hereby declare that all the work done in this project titled ”Backtracking Algorithm: To Buy or Not to Buy” is of our independent effort as a group.

6 Signatures

We hereby declare that all the work done in this project titled ”Backtracking Algorithm: To Buy or Not to Buy” is of our independent effort as a group.

吴杰枫

黄文杰

秦嘉俊

A Source Code (if required)

At least 30% of the lines must be commented. Otherwise the code will NOT be evaluated.

```
#include <iostream>
#include <vector>
#define MAXN 105
#define MAXT 62
#define INF (1<<30)
using namespace std;
bool flag = false;  /* indicates whether we have found a answer. */
int N, ans, min_extra = INF, min_left = INF;
vector<int>a[MAXN];
int Hash[MAXN][MAXT], target_beads[MAXT];  /* target_beads, Hash[k] are used to
record amounts of beads with the same color in the string. */
/*
Here we transfer char to int so that we can use it as the index of array.
(Obviously we can use the container map, but it will spend much time so we give up it.
*/
inline int CharToInt(char ch)
{
    if (ch >= '0' && ch <= '9') return ch - '0';     /* 0~9 */
    if (ch >= 'a' && ch <= 'z') return ch - 'a' + 10;   /* 10~35 */
    if (ch >= 'A' && ch <= 'Z') return ch - 'A' + 36;   /* 36~61*/
//      return -1;
}
/*
k - the k-th string

extra - The extra strings that we need to buy to provide the beads that Eva needs.
We need to find the least number of extra beads Eva has to buy when answer is YES.

left - How many beads that Eva still need to buy more strings to obtain them. We need
to find the least number of beads missing from all the strings when answer is NO.
*/
void DFS(int k, int extra, int left)
{
    if (extra >= min_extra) return;  /* Pruning 1: The "extra" must be nondecresing.
    So if the current "extra" has been bigger than the existing answer, then there is
    no need to search this branch further. */

    if (!left)      /* Pruning 2: left=0 indicates that we have gotten all beads she
    needs in the previous strings. Then there is no need to go continue. */
    {
        flag = true;
        min_extra = extra;
        return;
    }

    if (k == N)     /* We have reached the Boundary. */
    {
        if (min_left > left)    /* We want the left beads as small as possible. */
            min_left = left;
        return;
    }
    int save_extra = extra, save_left = left;
    /* To buy */
    bool bj = false;
    int i, delta[MAXT] = {0};    /* To record the changed amounts of the beads, so that
    we can restore them when backtracking. */
    for (auto i : a[k]) {
        if (target_beads[i]) {
            if (Hash[k][i] >= target_beads[i]) {    /* If for this color, the string
            have more beads than Eva needs, then the more beads will be extra beads. */
                delta[i] = target_beads[i];
                left -= delta[i];
                extra += Hash[k][i] - target_beads[i];
            }
            else {      /* If for this color, the string have less beads than Eva needs,
            then we just need to use up all beads. */
                delta[i] = Hash[k][i];
                left -= delta[i];
            }
            bj = true;      /* bj is used to indicate whether this string can cut down
            the amount that we still need to buy. Designed for Pruning 3.*/
        }
        else {
            extra += Hash[k][i];
        }
    }

    /* Pruning 3: If buying this string will have no benefit on the beads that Eva want
    to buy, then we will not search further again. Since the effect to buy is the same
    as not to buy. */
    if (bj) {
        /* Suppose we choose this string, then we can modify the beads that we still
        need to buy. */
        for (i = 0; i < MAXT; i++) {
            target_beads[i] -= delta[i];
        }

        DFS(k+1, extra, left);

        /* Backtracking, restoring to the state before we modify. */
        for (i = 0; i < MAXT; i++) {
            target_beads[i] += delta[i];
        }
    }

    /* Pruning 4: Modify the searching order: first we suppose to buy this string, then
    we search for the situation that we don't buy it. The advantage of the modified
    order is that if the answer is in the shallow layer of searching tree, we needn't
    spend much time useless branches. */

    /* Not to buy */
    DFS(k+1, save_extra, save_left);

    return;
```

```
}
int main()
{
    string target, s;
    cin >> target >> N;
    int i, j, len;
    len = target.length();
    for (i = 0; i < len; i++)
        target_beads[CharToInt(target[i])]++;   /* record amounts of beads with the
        same color in the Eva's string. */

    for (i = 0; i < N; i++) {
        cin >> s;
        len = s.length();
        for (j = 0; j < len; j++) {
            Hash[i][CharToInt(s[j])]++; /* record amounts of beads with the same
            color in one string. */
        }
        for (j = 0; j < MAXT; j++)
            if (Hash[i][j]) a[i].push_back(j);
    }
    double st = clock();    /* Record the start time. */
    DFS(0, 0, target.length());
    if (flag) cout << "Yes " << min_extra << endl;
    else cout << "No " << min_left << endl;
    double ed = clock();    /* Record the end time. */
    cout << "TIME: " << (ed - st) / CLOCKS_PER_SEC << endl;
    return 0;
}
```