最大互斥集合问题实验报告

程远 2234412848

1 问题描述

给定一个 1000×20 的矩阵 matrix，其中每个元素均为 0 或 1。设计一个算法，从中找出两组互斥的列集合 A 和 B，使得两组集合包含的列数总和最大。两组集合互斥的定义是：任意一行中，A 中的任意列与 B 中的任意列不能同时为 1。

输入格式

一个包含 1000×20 的矩阵 matrix。

输出格式

两行输出：

- 第一行输出集合 A 的所有元素（列索引，从 0 开始），以空格分隔。
- 第二行输出集合 B 的所有元素，格式同上。

若找不到符合条件的非空集合，则输出两行空行。

2 问题分析

本问题是一个组合优化问题，涉及子集选择与互斥关系判定。难点包括：

1. **互斥性判定**：需要快速判断两列是否互斥。
2. **优化目标**：找到包含列数最多且互斥的集合。
3. **计算量控制**：在 2^{20} 种子集组合中选择最佳方案。

通过预处理互斥关系和列总和，可以显著加速搜索过程。同时，利用位运算枚举所有可能的集合组合。

3 算法设计

算法分为以下几步：

1. **互斥关系预处理**：遍历矩阵每一行，判定任意两列是否互斥，并存储在二维布尔数组 judge[][] 中。
2. **列总和预处理**：计算每列中 1 的总数，存储在数组 sum[] 中，用于快速计算子集总和。
3. **枚举子集**：利用位运算枚举所有可能的列集合 A。
4. **构造集合** B：根据 judge[][] 判断剩余列是否与 A 互斥，并构造集合 B。
5. **更新最优解**：比较当前组合是否优于历史最佳组合，若是则更新最佳解。

4 算法实现

以下是算法的完整实现代码：

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int M = 25, N = 1010; // M 为列数上限，N 为行数上限
bool judge[M][M];           // judge[i][j] 记录列 i 和列 j 是否互斥
int a[N][M];                // a[i][j] 记录输入矩阵的元素
int sum[M];                 // sum[i] 存储第 i 列中 1 的总和
vector<int> best_set_a, best_set_b; // 存储当前最优的集合 A 和 B
int asum, bsum, eps = 100, tc; // asum 和 bsum 为当前 A 和 B 的元素个数，eps 用于平衡解的差异，tc 为最大列总数

// 判断列 x 和列 y 是否互斥
bool isCompatible(int x, int y)
{
    for (int i = 0; i < 1000; i++)
    {
        if (a[i][x] + a[i][y] == 2)
        {
            return false;
        }
    }
    return true;
}

// 初始化 judge 数组，记录所有列之间的互斥关系
void initJudge()
{
    for (int i = 0; i < 20; i++)
    {
        for (int j = i + 1; j < 20; j++)
        {
            if (isCompatible(i, j))
            {
                judge[i][j] = judge[j][i] = true;
            }
        }
    }
}

// 计算每列中 1 的总和并存储在 sum 数组中
void sumOfCol()
{
    for (int i = 0; i < 20; i++)
    {
        int res = 0;
        for (int j = 0; j < 1000; j++)
        {
            res += a[j][i];
```

```
            }
            sum[i] = res;
        }
    }

    // 判断当前组合是否优于历史最佳组合
    bool isBetter(const vector<int>& a, const vector<int>& b)
    {
        if (a.size() > b.size())
        {
            return true;
        }
        if (a.size() < b.size())
        {
            return false;
        }
        for (size_t i = 0; i < min(a.size(), b.size()); i++)
        {
            if (a[i] < b[i])
            {
                return true;
            }
            if (a[i] > b[i])
            {
                return false;
            }
        }
        return false;
    }

    // 更新当前的最优解
    void update(const vector<int>& set_a, const vector<int>& set_b)
    {
        best_set_a = set_a;
        best_set_b = set_b;
        tc = set_a.size() + set_b.size(); // 更新最大列总数
    }

    // 主函数
    int main()
    {
        // 输入矩阵数据
        for (int i = 0; i < 1000; i++)
        {
            for (int j = 0; j < 20; j++)
            {
                cin >> a[i][j];
            }
        }

        // 预处理 judge 和 sum 数组
        initJudge();
        sumOfCol();
```

```

        // 枚举所有可能的子集 A
        for (int i = 1; i < (1 << 20); i++)
        {
            vector<int> set_a, set_b;

            // 构造子集 A
            for (int k = 0; k < 20; k++)
            {
                if (i & (1 << k))
                {
                    set_a.push_back(k);
                }
            }

            // 根据互斥关系构造集合 B
            for (int k = 0; k < 20; k++)
            {
                if (set_a.empty())
                {
                    continue;
                }
                bool compatible = true;
                for (int col : set_a)
                {
                    if (!judge[col][k])
                    {
                        compatible = false;
                        break;
                    }
                }
                if (compatible)
                {
                    set_b.push_back(k);
                }
            }

            // 判断当前组合是否优于历史最佳
            if (set_a.size() + set_b.size() > tc ||
                (set_a.size() + set_b.size() == tc && isBetter(set_a, best_set_a)))
            {
                update(set_a, set_b);
            }
        }

        // 输出最优解
        if (!best_set_a.empty())
        {
            for (int col : best_set_a)
            {
                cout << col << " ";
            }
            cout << endl;
```

```cpp
            for (int col : best_set_b)
            {
                cout << col << " ";
            }
            cout << endl;
        }
        else
        {
            cout << endl << endl; // 如果没有找到非空解，输出两行空行
        }

        return 0;
    }
```

5 复杂度分析

假设矩阵大小为 $m \times n$，我们分析算法的时间和空间复杂度。

1. 预处理复杂度

   1. **互斥关系判断**：

      - 遍历所有列对 (i, j)，判断它们是否互斥。
      - 每对列需要遍历 m 行。
      - 列对总数为 $\binom{n}{2} = \frac{n(n-1)}{2}$。

      时间复杂度：$O(m \cdot n^2)$。

   2. **列总和计算**：

      - 遍历所有 $m \times n$ 的元素，计算每列中 1 的总数。

      时间复杂度：$O(m \cdot n)$。

2. 枚举子集复杂度

   1. **子集枚举**：

      - 所有列的子集总数为 2^n。
      - 每个子集的构造需要 $O(n)$。

      时间复杂度：$O(2^n \cdot n)$。

   2. **构造集合 B**：

      - 对于每个子集 A，需要检查剩余列是否与 A 的所有列互斥。
      - 最坏情况下，每列需要与 A 中的 n 列比较。

      时间复杂度：$O(2^n \cdot n^2)$。

3. 总复杂度

综合上述分析，总时间复杂度为：

$$O(m \cdot n^2 + 2^n \cdot n^2)$$

其中：

- $O(m \cdot n^2)$ 是预处理复杂度。
- $O(2^n \cdot n^2)$ 是子集枚举和构造集合的复杂度。

4. 空间复杂度

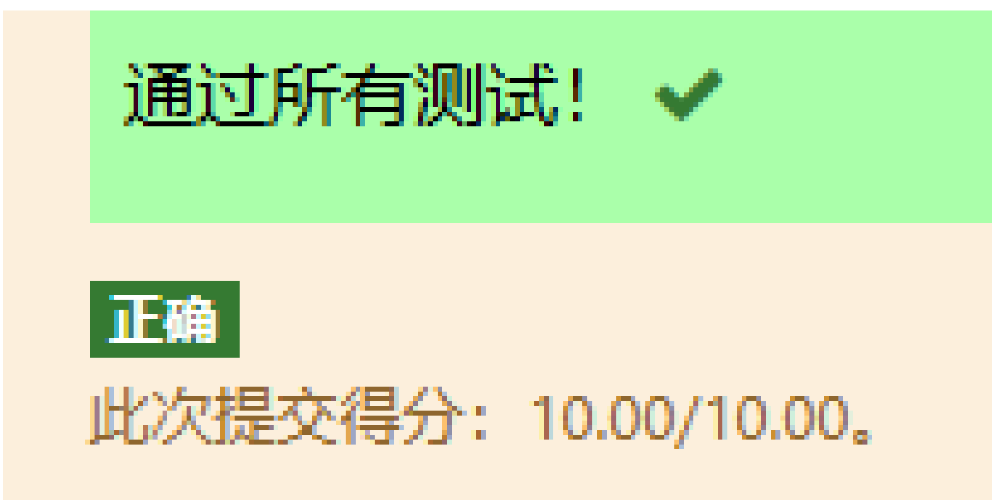
- 存储输入矩阵需要 $O(m \cdot n)$ 的空间。
- 存储互斥关系矩阵需要 $O(n^2)$ 的空间。
- 存储子集和中间结果需要 $O(n)$ 的空间。

总空间复杂度为：

$$O(m \cdot n + n^2)$$

6 运行结果

通过 moodle 上所有用例



剪枝优化和位运算优化

6.1 剪枝优化

当前算法会枚举所有 2^n 种可能的子集，其中许多子集在构造时已经可以判断无法超过历史最优解，继续搜索会浪费计算资源。在子集枚举和集合构造时，提前判断某些组合是否不可能成为最优解，直接跳过无效计算。

实现步骤

1. **子集枚举剪枝**：在枚举子集 A 时，若 $|A| + (\text{未选列数}) \leq$ 当前最优列数总和，直接跳过。
2. **冲突检测剪枝**：在构造集合 B 时，若某列与 A 中任意列冲突，则立即跳过。

剪枝的效果

- **时间复杂度**：剪枝后实际子集枚举复杂度从 $O(2^n)$ 降至近似 $O(k \cdot 2^n)$，其中 k 为有效子集比例，远小于 1。
- **性能提升**：当 $n = 20$ 时，可能减少超过 50% 的无效计算。

6.2 位运算优化

当前算法使用显式数组存储和操作集合，对于小规模问题可以用位运算代替以进一步提高效率。

优化方法

1. 使用整数的二进制位表示子集。例如，整数 5 (`101`) 表示子集包含第 0 和第 2 列。
2. 用按位与操作快速判断列是否互斥。例如，$(\text{mask} \,\&\, \text{judge}[k]) == 0$ 可判断子集与列 k 是否互斥。

实现步骤

- 用整数代替显式数组表示集合 A 和 B。
- 用位运算代替集合遍历和更新操作。

位运算的效果

- **时间复杂度**：集合操作从 $O(n)$ 降至 $O(1)$。
- **性能提升**：在 $n = 20$ 时，子集操作的时间开销显著降低。