锦标赛问题的两种解法

叶文霆 2014211519 2014211314

一、问题描述

设有n个运动员要进行网球循环赛。设计 一个满足下列条件的比赛日程表:

- 每个选手必须与其他n-1个选手各赛一次;
- 每个选手一天只能赛一次;
- 当n是偶数时,循环赛进行n-1天。
- 当n是奇数时,循环赛进行n天。

二、初步分析

首先我们先来考虑一下该问题的特殊版本，即当 N = 2^k 时，我们是怎样解决这个问题的:

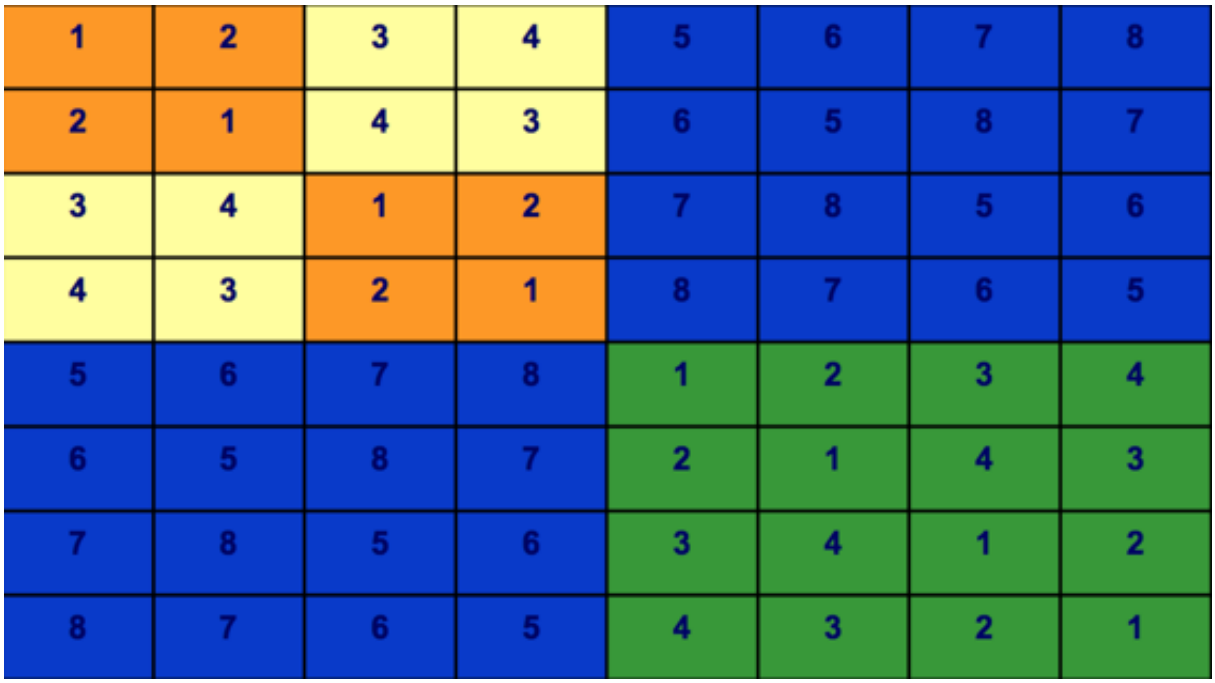

figure 1. N = 8 的解决方案

对于每一个size = n的问题，考虑到 n 是2的幂次，我们可以把问题划分为**完全相同**的两个子问题，然后再将 n / 2 的赛事安排**映射**到另一半的问题上。怎么映射呢？我们将 size = n 的的赛事分为两部分：

（出于方便，令 sub_group = n / 2 ）

1. 分别发生在其子空间**内部**的比赛。我们通过调用Make_Schedule(sub_group)来安排前一半的比赛。那么与此同时，我们怎么安排另一半的比赛呢？**直接把左边对应下标 x + n / 2就可以了！**由于每个人需要和 sub_group - 1 个人比赛，所以内部比赛需要 sub_group - 1的时间（在这里存在第一行，即每个选手和自己打，是为了后面考虑，实际中不输出该行）。表达式如下：

```
date[i][j] = date[i][j - sub_group] + sub_group;
```

因为在左半部分我们已经安排了一个合法的比赛安排，我们通过一个 one-to-one 映射到右半部分的对应点上，其安排一定也是合法的。

2. 分别发生在其子空间之间**交叉**的比赛。现在我们同样使用熟悉的映射函数 **y = x + sub_group**,与步骤 1 不同之处在于其把原来 安排给 选手 i 的对手改为date[i][j] + sub_group，即其内部对手在另一小组的映射。由于每个人需要和另一个组的每个人都打比赛，所以一共需要 sub_group 天比赛。表达式如下：

```
date[i][j] = date[i - sub_group][j] + sub_group;
```

从这里我们就知道，在内部比赛的时候每个选手不能和自己打，**但是在子空间之间的比赛，每个选手是要和自己映射的那个选手比赛的**，我们之前所说的添加了“无意义”的第一行，正是为了此处处理的方便。

但是当 N 不再是2的幂次时，问题将新增了以下难题：

- 每次平分子问题时，两遍规模不一定相同
- 人数为奇数/偶数，所需要的天数不一样，我们又该如何合并子问题？

三、数据结构设计

```
class schedule
{
class schedule
{
public:
    schedule():date(0),compete(0){}
    schedule(int);
    schedule(std::ifstream &);
    std::vector< std::vector<int> >   date;          //存放时间安排表
    std::vector < std::vector<bool> > compete;       //compete[i][j]
表示 i 和 j是否打过比赛
    bool success;
    int n, m;                                        // n 是人数 m 是
所要的天数
    void Make_Schedule(const int &);
    int Get_Duration(const int &);
    void Print_Schdule();
};
```

四、算法设计

1.暴力DFS+回溯

初遇问题毫无头绪，笔者就顺手写了一个深搜加上回溯的算法，用了 Model 的基本性质：

1. 如果问题规模为偶数，则每一天每个人都不能空闲。
2. 如果问题规模为奇数，则每个人**有且仅有**一天空闲。

利用这个性质，我们可以写一个朴素的搜索，代码如下：

```
void Naive_Algorithm(const int &n)
{
    schedule table(n + 1);                                  //新建赛程表
    if (n % 2 == 0)
        for (int day = 1; day <= n-1; ++day)                //n-1 days
        {
            table.success = 0;
            Arrange_Game(day, 1, table, 0);
        }
    else
        for (int day = 1; day <= n; ++day)                  //n days
        {
            table.success = 0;
            Arrange_Game(day, 1, table, n - day + 1);
        }
}
```

```
void Arrange_Game(const int day, const int x, schedule &table, cons
t int &spare)
{
    int n = (int) table.date.size() - 1;
    if (x > n)
    {
        table.success = 1;
        return;
    }

    if (table.date[day][x] != 0 || x == spare)
        Arrange_Game(day, x + 1, table, spare);
    else for (int i = x + 1; i <= n; ++i)
        if (table.compete[x][i] == 0 && i != spare && table.date[da
y][i] == 0)
        {
            table.date[day][x] = i; table.date[day][i] = x;
            table.compete[i][x] = 1; table.compete[x][i] = 1;
            Arrange_Game(day, x + 1, table, spare);
            if (table.success == 1) return;
            table.date[day][x] = 0; table.date[day][i] = 0;
            table.compete[i][x] = 0; table.compete[x][i] = 0;
        }
}
```

当问题规模是奇数的时候，每次选举出一个空闲的人，然后用深搜给每个人**合法地**安排一个对手。其中深搜的搜索树大小理论上是 O(N!)级别，所以经测试，这个算法只能够适用于 N 较小（$N \leqslant 26$）的情况下。

2. Divide & Conquer

千呼万唤始出来，我们今天的主角终于出现了。能够用“分而治之”的思想来解决问题的前提是，这个问题可以划分为若干个子问题。当问题的规模是一个奇数的时候，我们需要借助一个**哨兵**来实现我们的算法。我们举个例子来介绍一个哨兵：

```
N = 6
2 1 4 3 6 5
3 5 1 6 2 4
4 6 5 1 3 2
5 4 6 2 1 3
6 3 2 5 4 1

N = 5
2 1 4 3 0
3 5 1 0 2
4 0 5 1 3
5 4 0 2 1
0 3 2 5 4
```

上图是使用爆搜生成出来的结果，现在我们这么考虑：处理 N = 5的时候，我们假装存在一个编号的6的哨兵，然后就可以将问题转换为一个规模为偶数的问题来进行求解。求解出来以后，我们只需要再将哨兵删掉就可以了。

在上图中的行数代表天，列数代表人。如果我们将 N = 6 时间表，找到每一行第六列的数据 i，将这一行的第 i 列设为0，最后再把第六列删掉，*我们就得到了 N = 5的时刻表*。通过这个方法，我们每一步就可以将问题转化为一个偶数规模进行求解，这样我们就能够进行平均划分了。然而另一个问题迎面而来：**“如果两个子问题的规模是奇数，那么整体就会有两个哨兵，该怎样合并子问题呢？”**

我们采用同样的思路，将比赛分为内部比赛和交叉比赛：

1. 内部比赛，共sub_group天。由前面的分析我们可以知道，规模为奇数的子问题中，**每个人有且仅有一次被轮空**。所以每一天我们为左右两个部分被轮空的人安排比赛。代码如下：

```
//内部比赛
for(int i = 1; i <= duration; i++)
    for (int j = sub_group + 1; j <= 2 * sub_group; j++)
        if (date[i][j - sub_group] == 0)
        {
            date[i][j - sub_group] = j;
            date[i][j] = j - sub_group;
        }
        else date[i][j] = date[i][j - sub_group] + sub_group;
```

2. 交叉比赛，共 sub_group - 1 天。对于编号为 i 的选手，在内部比赛中由于轮空一次，他已经和编号为 sub_group + i 的选手打过比赛了。接下来的 sub_group - 1 天，只要和另外一组的 sub_group - 1 个人都打过一次比赛就好了，在这里我们采用如下的**平移对应**方法来安排比赛，即交叉比赛的第 k天，下标为 i($i \leqslant sub\_group$)的选手，安排其和编号为 sub_group + i + k 选手比赛（如果超界了就需要取 模）。代码如下：

```
//两个 subgroup 互打
k = (sub_group % 2 == 1)? 1 : 0;
for (int i = duration + 1; k < sub_group; k++, i++)
    for (int j = 1; j <= sub_group; j++)
    {
        int next;
        if ((j + k) % sub_group == 0) next = 2 * sub_group;
        else next = sub_group + (j + k) % sub_group;
        date[i][j] = next;
        date[i][next] = j;
    }
```

这里的 “k = (sub_group % 2 == 1)? 1 : 0 ”是考虑了子问题规模是偶数的情况：偶数人数的情况下由于没有和另一小组对应的选手比赛过，所以偏移量初始值为0。

经过以上两步，我们就能够讲一个 sub_group 的解扩展到整个问题，从而实现分治算法。需要的话，我们再讲问题的规模减小一，就能够解决任意规模的问题了。

该 Divide & Conquer 的**完整代码**见下一页（已封装到 schedule 类中）：

```
void schedule:: Make_Schedule(const int &n)
{
    if (n == 2)
    {
        date[1][1] = 2;
        date[1][2] = 1;
        return;
    }
    int sub_group = (n + 1) / 2, duration = Get_Duration(sub_grou
p);
    Make_Schedule(sub_group);
    //内部比赛
    for(int i = 1; i <= duration; i++)
        for (int j = sub_group + 1; j <= 2 * sub_group; j++)
            if (date[i][j - sub_group] == 0)
            {
                date[i][j - sub_group] = j;
                date[i][j] = j - sub_group;
            }
            else date[i][j] = date[i][j - sub_group] + sub_group;
    //两个 subgroup 互打
    int k = (sub_group % 2 == 1)? 1 : 0;
    for (int i = duration + 1; k < sub_group; k++, i++)
        for (int j = 1; j <= sub_group; j++)
        {
            int next;
            if ((j + k) % sub_group == 0) next = 2 * sub_group;
            else next = sub_group + (j + k) % sub_group;
            date[i][j] = next;
            date[i][next] = j;
        }
    //delete the fake people
    if (n % 2 == 1)
        for (int i = 1; i <= Get_Duration(n); ++i)
            date[i][date[i][2 * sub_group]] = 0;
}
```

五、正确性与性能

A.正确性验证

为验证程序的正确性，我编写了一个check_rightness.cpp，它能够从当前目录读取”Schedule.txt“，其中有问题的规模与赛事安排表。如果满足要求则输出 Correct，代码如下：

```
void Quit(int x)
{
    if (x == 1)
        cout << "The answer is correct." << endl;
    else
        cout << "The answer is wrong:(" << endl;
    exit(0);
}

int main(int argc, const char * argv[])
{
    ifstream infile("Schedule.txt");

    schedule table(infile);
    for (int day = 1; day <= table.m; ++day)
    {
        vector <bool> used(table.n + 1);
        for (int i = 1; i <= table.n; ++i)
            if (used[table.date[day][i]] == 0 && table.compete[i][t
able.date[day][i]] == 0)     // new and spare opponent
            {
                used[table.date[day][i]] = 1;
                table.compete[i][table.date[day][i]] = 1;
            }
            else
                Quit(0);
    }
    Quit(1);
    return 0;
}
```

对于朴素的搜索+回溯算法，我对 N = 2，5，8，11…23 的每一种情况都运行了测试程序，结果均正确。

对于第二种分治算法，我取 N = 2, 10, 100, 200, 500, 1000, 2000, 5000, 7000, 10000 共10个点运行了检验程序，结果均正确。

B. 理论时间复杂度

第一种算法由于需要用在 O(n!)的状态空间里面搜索合法比赛安排，故总的时间复杂度应该在 O(n*n!)。
第二种算法中，我们将一个规模为 n 的为题化为一个 n / 2 规模的问题，还用 O(n^2)的额外时间进行安排表的填充，可得时间复杂度关系式：**T(n) = T(n/2) + O(n^2)**

通过运用Master Method，计算得出其时间复杂度为 O(n^2)。由于输出的时刻表规模就是 n^2，可知不存在比起更加高效的算法了。

C. 实际时间消耗

通过批处理的手段，我们可以通过 Mathematics作图函数，我们可以更加直观的观察二者效率的区别

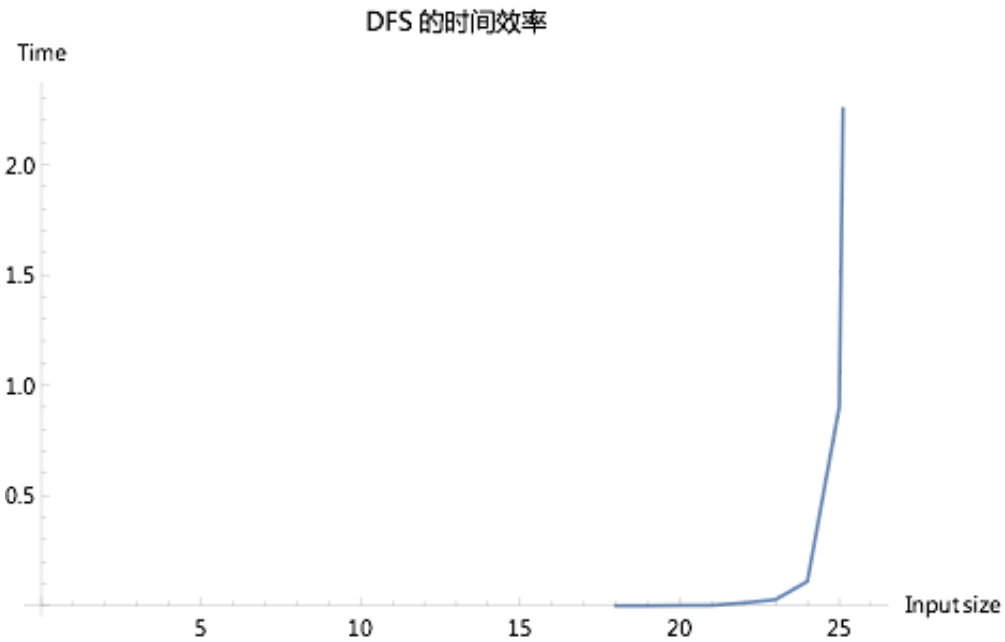

figure 2. DFS 的实际时间复杂度

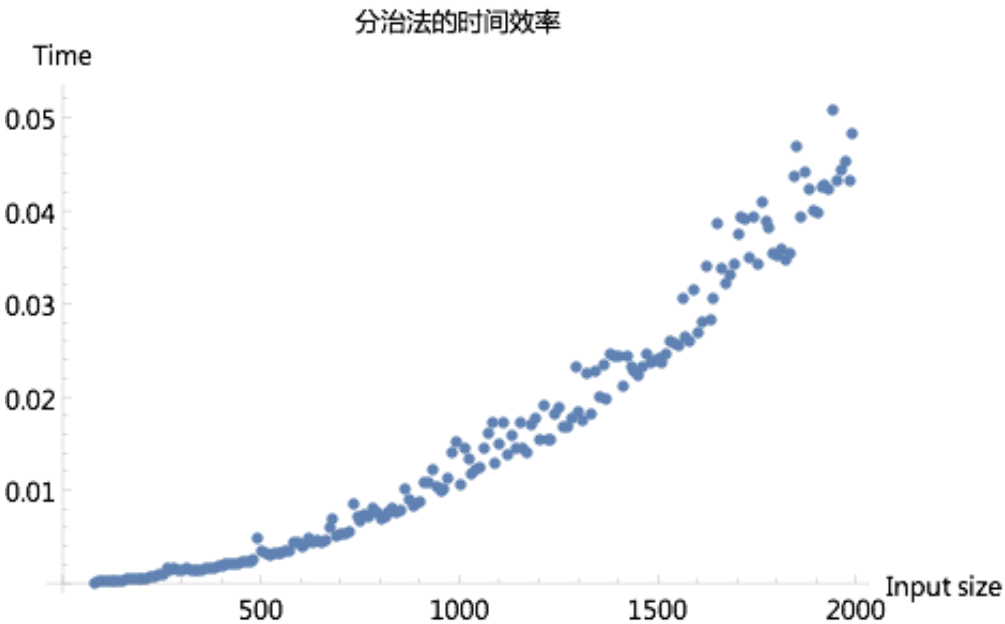

figure 3. 分治法的实际时间复杂度

figure 2说明 DFS 的时间复杂度增长速度非常惊人，在 N = 25左右时开始爆发，作证了我们理论推测的 O(n*n!)时间复杂度。而分治算法（见 figure 3）的效率则好得多，在 N = 2000的时候消耗时间仅为0.05s，其图像也近似为 O(N^2) 。

经过分析，证明分治法确实能够有效地提高程序的性能和效率。