

# Lab 7: 最佳调度问题的回溯算法

PB21020718 曾健斌

## 一、实验内容

设有  $n$  个任务由  $k$  个可并行工作的机器来完成,完成任务  $i$  需要时间为  $t_i$ 。试设计一个算法找出完成这  $n$  个任务的最佳调度,使完成全部任务的时间最早 (要求给出调度方案)。

### 程序输入

从 test 系列文件获取数据

第一行为任务数  $n$  和机器个数  $k$ , 第二行为完成任务  $i$  需要的时间  $t_i$ , 包含  $n$  个数据,以空格间隔

### 程序输入

输出三个测试案例所有任务完成的总时间, 及调度方案

## 二、算法实现

首先定义一个类, 设置类字段以减少反复传入参数等等操作的耗费

```
// optimalscheduling.h

class Optimal {
public:
    Optimal(vector<int> time, int n, int k) {
        this->n = n;
        this->k = k;
        this->cost = INT_MAX;           // 将费用初始化为 INT_MAX
        this->Time = time;
        this->match = vector<int>(n, -1); // 匹配初始化为全 -1 以表示未匹配
        this->bestmatch = vector<int>(n, -1);
        this->current = vector<int>(k, 0); // 当前各机器费用初始化为 0
    }
    ~Optimal() {}
    void optimalschedulingsub(int i);
    void optimalscheduling();
    int cost;           // 费用
    vector<int> bestmatch; // 任务与机器的最佳匹配

private:
    int n;
    int k;
    vector<int> match; // 任务与机器的当前匹配
    vector<int> current; // 当前各及其工作费用
    vector<int> Time; // 各任务耗费
};
```

定义四个数组以及 cost 变量以供回溯递归使用, 构造器使用 time 数组, 任务数以及机器数进行初始化

然后是 `optimalscheduling()` 与 `optimalschedulingsub()` 的实现：

```
// optimalscheduling.cpp

void
Optimal::optimalscheduling()
{
    optimalschedulingsub(0);
}

void
Optimal::optimalschedulingsub(int i)
{
    int flag;
    int tmp = *max_element(current.begin(), current.end());
    if (cost <= tmp)
        return;
    if (i == n) { // 边界条件
        cost = tmp;
        bestmatch = match;
        return;
    }
    for (int j = 0; j < k; j++) {
        flag = 1;
        for (int l = 0; l < j; l++) {
            if (current[l] == current[j]) {
                flag = 0;
                break;
            }
        }
        if (!flag || current[j] + Time[i] >= cost)
            continue;
        current[j] += Time[i]; // 选择
        match[i] = j;
        optimalschedulingsub(i + 1); // 递归进入下一层
        current[j] -= Time[i]; // 撤销选择 (回溯)
    }
}
```

此处只考虑回溯框架，剪枝策略在后面单独讲。

递归的边界条件为层数达到  $n$ 。选择列表为 `current`，当前位置为 `Time[i]`

### 三、剪枝策略

```
void
Optimal::optimalschedulingsub(int i)
{
    int flag; // 1
    int tmp = *max_element(current.begin(), current.end());
    if (cost <= tmp) // 2
        return;
    if (i == n) {
```

```

        // the following code will lower the performance quite a lot      // 5
        // if(tmp <= cost){
        //     cost = tmp;
        //     bestmatch = match;
        // }
        cost = tmp;
        bestmatch = match;
        return;
    }
    for (int j = 0; j < k; j++) {
        flag = 1;
        for (int l = 0; l < j; l++) {      // 3
            if (current[l] == current[j]) {
                flag = 0;
                break;
            }
        }
        if (!flag || current[j] + Time[i] >= cost)      // 4
            continue;
        current[j] += Time[i];
        match[i] = j;
        optimalschedulingsub(i + 1);
        current[j] -= Time[i];
    }
}


```

// 2 处进行剪枝，在发现当前费用已经达到或者超过此前可以达到的最优费用时跳过该子树。值得注意的是本实验只要求找到一个可行的最优策略，所以剪枝时将当前最优策略的子树一并减去，测试表明，如果选择 // 5 的策略（即不剪去相等情况的子树），则算法运行时间将增长上百倍

// 3 处与 // 1 处是将 `current` 相等的机器视为等同，以减去重复情况

// 4 处跳过会导致费用超过最佳费用的选择

## 四、实验结果

 image-20231113135307276

正确输出三个测试的最佳调度方案以及算法耗时，三个测试均在 0.5 秒内通过

## 五、实验总结

本次实验主要任务是回溯算法的实现，主要难点在于剪枝策略的选择与取舍

- 深切地体会到了剪枝对于回溯搜索的性能提升
- 对于回溯与剪枝有了更多心得