Report For Scheduling Problem1

09019204
曹邹颖

1. Problem description / demand analysis 问题描述

Let E = {σ_1,…, σ_n} be a set of intervals such that for each interval σ_i is defined as follows $\sigma_i = [a_i, b_i)$ with $a_i \le b_i$. A schedule is feasible if for any two intervals σ_a, σ_b we have $\sigma_a \cap \sigma_b = \varnothing$. We consider the profit of a solution as the total length of the interval. If S is feasible, then the profit is $p(S) = \sum_{i \in S}(b_i - a_i)$.The goal is to find a feasible solution S such that p(S) is maximized.

For example, E = {[0,3), [1,4), [3,7), [8,10)}

S = {[0,3), [3,7)} is a feasible schedule with p(S) = (3 − 0) + (7 − 3) = 7

S = {[0,3), [1,4)} is not a feasible schedule.

1. Design a greedy algorithm to select the intervals according to the input
2. Is this algorithm optimal? If it is not, find a counter example (bonus: find the worst case of the algorithm)
3. Try different order of the inputs and run the greedy algorithm again
4. Design a dynamic programming algorithm to solve this problem

2. System structure / algorithm idea 算法设计

(1) basic idea 基本思想

贪心算法：将 feasible solution 看作一个栈 S，对于输入的 intervals 序列，依次判断，是否和 S 栈顶的区间重叠，若不重叠则该区间入栈 S。

动态规划：将输入的 intervals 序列按照其右端点从小到大排序，用 profit[i]表示前 i+1 个区间的最佳 feasible solution 利润，对于区间 i 只有两个选择，①选择，找到与区间 i 不重叠的最大序号 j，其利润为 profit[j]+区间 i 利润，②不选择，其利润就为 profit[i-1]，这样便构造好动态规划算法的核心状态转换方程。

(2) system framework 代码框架

贪心算法：由题意，无需对输入的 intervals 序列进行排序，依次判断即可。主要代码核心便是看当前 feasible solution 的栈 S 顶的区间右端点可用 S_end 进行标记，看带判断的区间左端点是否>=S_end，若满足>=即可加入栈 S 中。

动态规划：采用快速排序算法 void QuickSort(int* end,int* start, int x, int y)将 end 数组(区间右端点)从小到大排序，同时对应的 start 数组(区间左端点)相应变换，之后需利用一个数组 p 记录排好序的 intervals 序列中每一个区间 i 之前，可以取到的与 i 不重叠的区间最大序号，最后代码核心即为状态转换方程：

```
if (p[i] != -1)
    profit[i] = max(profit[i-1], profit[p[i]] + (end[i] - start[i]));
else
    profit[i] = max(profit[i-1], end[i] - start[i]);
```

profit[i]表示前 i+1 个区间的最佳 feasible solution 利润。

最佳 feasible solution 结果打印部分，用 vector<vector<int> >S 数据结构，vector S[i]记录前 i+1 个区间的最佳 feasible solution 结果。这样在状态转换时将选择的 intervals 添加入每个对应的 vector 中记录即可。

(3) the functions and relationships of each module 模块的功能和关系

贪心算法：可以按序输入 intervals 的左右端点时，便判断该区间是否可以加入 feasible solution 中，看区间左端点是否>=当前 feasible solution 的栈 S 顶的区间右端点 S_end，若满足>=即可加入栈 S 中；

动态规划：最佳 feasible solution 中的 intervals 需要在每次更新 profit 时记录下来，即当 p[i]≠-1 时，如果 profit[i - 1] > profit[p[i]] + (end[i] - start[i])则 feasible solution S[i]=S[i - 1]否则 feasible solution S[i]中需加入 S[p[i]]中所有区间+区间 i 本身，当 p[i] = -1 时，如果 profit[i - 1] > (end[i] - start[i])则 feasible solution S[i]=S[i - 1]否则 feasible solution S[i]即为区间 i 本身，而打印最佳 feasible solution 部分是求解最佳 feasible solution 模块分开的。

3. Function module design 功能模块设计

(1) function module design idea

贪心算法：

for 所有待选择的 interval i
    按序输入 interval i 的左右区间(start,end)
    查看当前 feasible solution 的栈 S 顶的区间右端点 S_end
    如果 start>=S_end，则区间 i 选入 feasible solution 栈 S 中

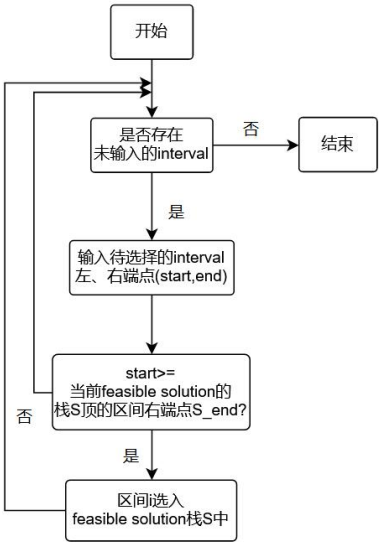
动态规划：

求解最佳 feasible solution 模块：
Quick Sort(end,start,0, n-1);将输入的 intervals 序列按照右端点从小到大排序
for 所有的 interval i
    找到每一个区间 i 之前，可以取到的与 i 不重叠的区间最大序号 p[i]
for 所有的 interval i
    如果 p[i]!= -1:区间 i 之前，存在不重叠的区间
        profit[i] = max(profit[i - 1],profit[p[i]] + (end[i] - start[i]));
    否则，区间 i 之前，不存在不重叠的区间
        profit[i] = max(profit[i - 1],end[i] - start[i])

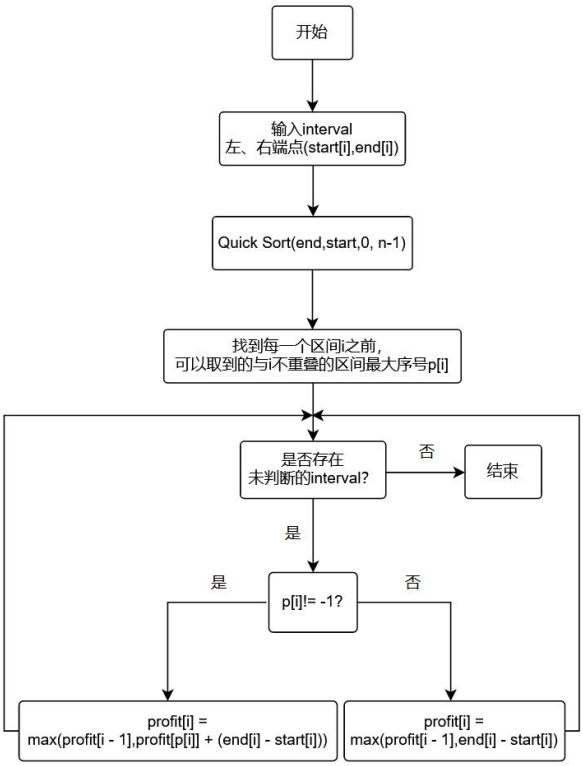
最佳 feasible solution 打印模块：
(用 vector[n-1]记录的 n 个 intervals 选择的最佳 feasible solution)：
for i=0...S[n-1].size()
    输出 start[S[n-1][i]]以及 end[S[n-1][i]]

(2) flow chart

贪心算法：



动态规划：



(3) algorithm complexity analysis

贪心算法：O(n)，n:区间个数；

动态规划：找每一个区间 i 之前，可以取到的与 i 不重叠的区间最大序号 p[i]时最坏可能达到 $O(n^2)$，而状态转移求解最佳 feasible solution 的 profit 数组时(不算记录最佳 feasible solution 解法)算法复杂度为 O(n)。

4. Test results and exercises answers 测试结果和练习回答

(1) test data selection or generation method 测试数据选择/生成方法

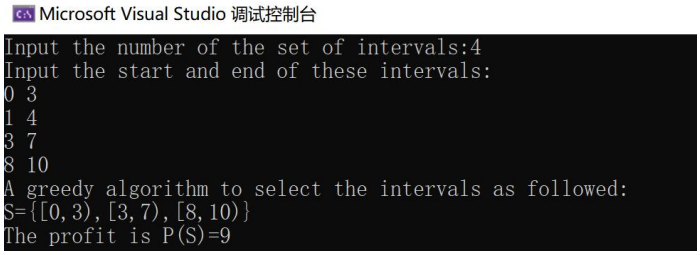
Data set

- E = {[0,3), [1,4), [3,7), [8,10)}
- E = {[8,10), [1,3), [4,5), [6,7), [9,15), [1,10), [5,8)}
- E = {[10,19), [13,23), [0,9), [4,19), [6,9), [7,12), [1,14), [15,35)}

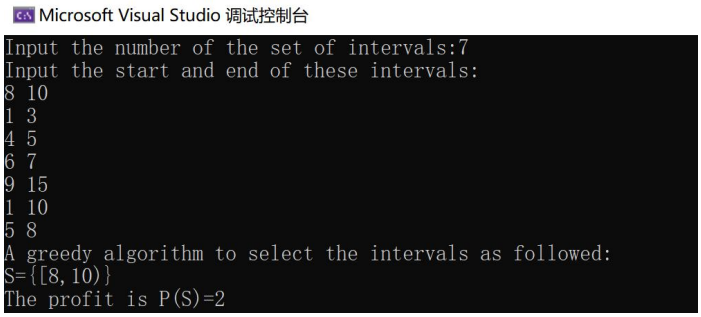
(2) operation result screenshot 运行结果截屏

贪心算法：

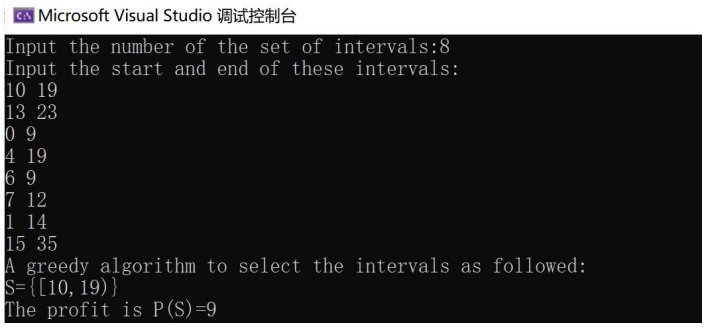
- E = {[0,3), [1,4), [3,7), [8,10)}



- E = {[8,10), [1,3), [4,5), [6,7), [9,15), [1,10), [5,8)}

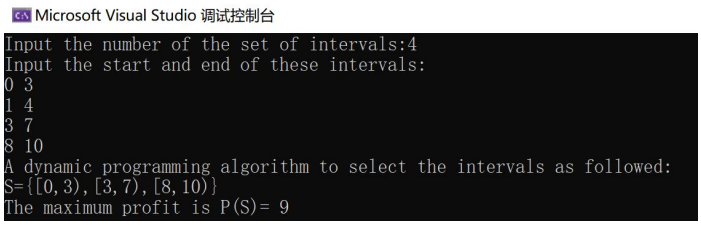


- E = {[10,19), [13,23), [0,9), [4,19), [6,9), [7,12), [1,14), [15,35)}

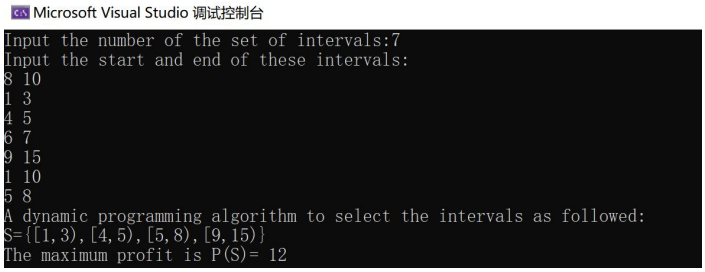


动态规划：

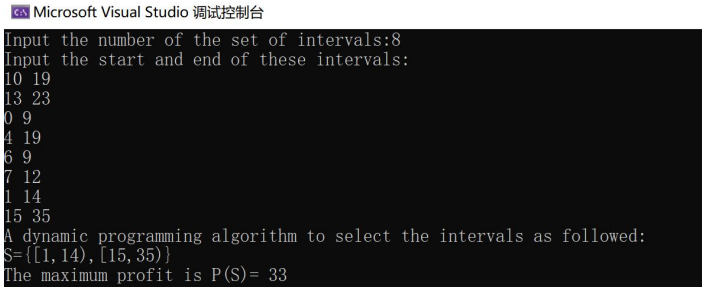
- E = {[0,3), [1,4), [3,7), [8,10)}



• E = {[8,10), [1,3), [4,5), [6,7), [9,15), [1,10), [5,8)}



• E = {[10,19), [13,23), [0,9), [4,19), [6,9), [7,12), [1,14), [15,35)}



(3) answers to the questions 问题的回答

1. 设计的贪心算法见 Report 6.
2. 贪心算法解决这问题不是最优的。

   反例：E = {[8,10), [1,3), [4,5), [6,7), [9,15), [1,10), [5,8)}

   贪心算法求得：

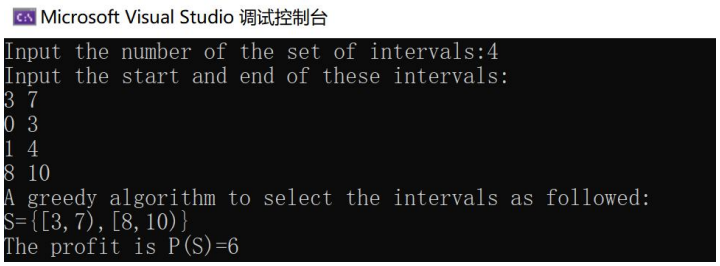
   S={[8,10)}, maximum profit=2

   而最佳的解决方案为：

   S={[1,3), [4,5), [5,8), [9,15)}, maximum profit=12

   该贪心算法的最坏情况：输入的第一个区间的右端点是所有区间端点中最大的一个，这样会导致将第一个区间选入 solution 中后续再无其他区间选入的可能。
3. 针对 E = {[0,3), [1,4), [3,7), [8,10)},

   改变输入顺序为{[3,7) ,[0,3), [1,4), [8,10)}，便得到不一样的解决方案如下：



4. 设计的动态规划算法见 Report 6.

5. Experimental summary 实验总结

(1) the problems encountered 遇到的问题

问题：

如何设计动态规划算法来得到最佳 feasible solution 是一个难题。

(2) the problem-solving process 解决方案

解决方案：

将输入的 intervals 序列按照其右端点从小到大排序，利用一个数组记录排好序的 intervals 序列中每一个区间 i 之前，可以取到的与 i 不重叠的区间最大序号，用 profit[i]表示前 i+1 个区间的最佳 feasible solution 利润，对于区间 i 只有两个选择，①选择，找到与区间 i 不重叠的最大序号 j，其利润为 profit[j]+区间 i 利润，②不选择，其利润就为 profit[i-1]，这样便构造好动态规划算法的核心状态转换方程。

(3) summarize the experimental experience 实验总结

实验中，体会了贪心算法总是选择当前最优的，期望通过局部最优选择来得到一个全局最优解，但是对于某些问题来说，一些贪心的选择并不能得到最优解，只有考虑动态规划算法才能得出最佳解决方案。

6. Source code 源代码

```
#include<iostream>
#include<vector>
using namespace std;
void QuickSort(int* a,int* b, int x, int y);    // 快速排序
void Swap(int& x, int& y);                      // 交换
void greedySelect();                            // 贪心算法选择（由题意无需排序）
int main()
{
    int n;                                      // 区间的个数
    cout << "Input the number of the set of intervals:"; cin >> n;
    int* start = new int[n];                    // 区间的左端点
    int* end = new int[n];                      // 区间的右端点
    vector<vector<int> >S;                      // 前 i+1 个区间的最佳 feasible solution,记录最优解路径
    for (int i = 0; i < n; i++)
        S.push_back(vector<int>());
    cout << "Input the start and end of these intervals:" << endl;
    for (int i = 0; i < n; i++)
        cin >> start[i] >> end[i];
    QuickSort(end, start, 0, n - 1);            // 按 end 从小到大排序
    int* profit = new int[n];                   // profit[i]:前 i+1 个区间的最佳 feasible solution 利润
    int* p = new int[n];                        // p[i]:区间 i 之前，可以取到的不重叠的区间序号
    for (int i = 0; i < n; i++){
        profit[i] = 0;                          // 初始化，0 表示不存在 feasible solution，利润为 0
        p[i] = -1;                              // 初始化，-1 表示不存在
        for (int j = i - 1; j >= 0; j--)
            if (end[j] <= start[i]) {
                p[i] = j;                       //使得区间 j 与 i 不重叠的最大序号,j<i
```

```
                    break;
                }
        }
        profit[0] = end[0] - start[0];          // 只有一个区间时，最佳 feasible solution 就是区间本身
        S[0].push_back(0);
        for (int i = 1; i< n; i++){             // 区间 i 两个选择:选择:找到与区间 i 不重叠的最大序号
                                                        j,profit[j]+区间 i 利润;不选择:profit[i-1]
            if (p[i] != -1){                    // 区间 i 之前，存在不重叠的区间
                if (profit[i - 1] > profit[p[i]] + (end[i] - start[i])){
                    profit[i] = profit[i - 1];// 不选区间 i 利润大
                    for (int j = 0; j < S[i - 1].size(); j++)
                        S[i].push_back(S[i - 1][j]);
                }
                else {                          // 选区间 i 利润大
                    profit[i] = profit[p[i]] + (end[i] - start[i]);
                    for (int j = 0; j < S[p[i]].size(); j++)
                        S[i].push_back(S[p[i]][j]);
                    S[i].push_back(i);
                }
            }
            else {                              // 区间 i 之前，不存在不重叠的区间
                if (profit[i - 1] > (end[i] - start[i])) {
                    profit[i] = profit[i - 1];// 选区间 i 利润大
                    for (int j = 0; j < S[i - 1].size(); j++)
                        S[i].push_back(S[i - 1][j]);
                }
                else {                          // 选区间 i 利润大
                    profit[i] = end[i] - start[i];
                    S[i].push_back(i);
                }
            }
        }
        cout << "A dynamic programming algorithm to select the intervals as followed:\nS={";
        for (int i = 0; i < S[n-1].size(); i++) {  // 打印最佳 feasible solution
            cout << "[" << start[S[n-1][i]] << "," << end[S[n-1][i]] << ")";
            if (i != S[n-1].size() - 1)cout << ",";
        }
        cout << "}\nThe maximum profit is P(S)= " << profit[n-1] << endl;
        delete[] start;
        delete[] end;
        delete[] profit;
        delete[] p;
        return 0;
}
```

```
void QuickSort(int* a,int* b, int x, int y)
{
	if (x >= y)return;
	int pivot = a[x], pivotpos = x;
	for (int i = x; i <= y; i++)
	{
		if (a[i] < pivot)
		{
			pivotpos++;
			if (pivotpos != i)		// 小于基准的交换到左侧
			{
				Swap(a[pivotpos], a[i]);
				Swap(b[pivotpos], b[i]);
			}
		}
	}
	Swap(a[x], a[pivotpos]);
	Swap(b[x], b[pivotpos]);
	QuickSort(a, b, x, pivotpos - 1);
	QuickSort(a, b, pivotpos + 1, y);
}
void Swap(int& x, int& y)
{
	int temp = x;
	x = y;
	y = temp;
}
void greedySelect()				// 贪心算法选择（由题意无需排序）
{
	int n;					// 区间的个数
	cout << "Input the number of the set of intervals:"; cin >> n;
	int* start = new int[n];		// 区间的左端点
	int* end = new int[n];			// 区间的右端点
	vector<int>S;				// 最佳 feasible solution,vector S 记录最优解路径
	int S_end = 0;				// 加入 S 中的最后一个区间的右端点
	int profit = 0;				// 最佳 feasible solution 利润
	cout << "Input the start and end of these intervals:" << endl;
	for (int i = 0; i < n; i++)
	{
		cin >> start[i] >> end[i];
			if (start[i] >= S_end) {	// 区间 i 的左端点<=S 中的最后一个区间的右端点
			profit += end[i] - start[i];
			S_end = end[i];
			S.push_back(i);		// 区间 i 可加入 S
```

```
            }
        }
        cout << "A greedy algorithm to select the intervals as followed:\nS={";
        for (int i = 0; i < S.size(); i++) {          //  打印最佳 feasible solution
            cout << "[" << start[S[i]] << "," << end[S[i]] << ")";
            if (i != S.size() - 1)cout << ",";
        }
        cout << "}\nThe profit is P(S)=" << profit << endl;
        delete[] start;
        delete[] end;
}
```