关于单机调度问题的贪心算法

王圣富

- 北京理工大学计算机学院 2016 级
- 学号：1120161848
- 班号：07111605

摘　要

针对工件不同释放时间、加工时间和实际完工时间的复杂情况，研究单机调度总延迟时间最小化问题。根据问题的 NP-HARD 特性，我们选择化简问题以实现近似满足贪心选择性质。本文涉及的三种贪心算法，利用了工件加工与当前时间的关系，降低了算法的时间复杂度。同时本文绘制了甘特图，更加直观地展现调度的过程。实验结果表明，与穷举算法相比，贪心算法在运行速度上有更好的表现，但是求解质量视具体情况而定。

关键词

单机调度；

贪心算法；

最小总延迟；

甘特图

§1 引言

调度在各行各业的广泛应用引起了人们的浓厚兴趣，在运筹学、应用数学、计算机科学、生产管理科学以及人工智能等领域，大量新的文献不断涌现。这其中，单机调度问题一直是研究的热点，许多复杂的调度问题可分解为多个单机调度问题来求解，深入研究单机调度问题可以更好地帮助人们求解复杂的多机调度问题，因此单机调度问题在理论研究和实际应用中有着很高的价值。

无可否认，单机调度问题是实际存在的，例如一台进行流程式生产的机器，每次只能生产一种产品，可以看作单机调度问题。对于有多台机器的工厂或车间，往往有一台瓶颈机，它的

作用的发挥，会直接影响到全厂产品的生产。这也是可以按单机调度问题来考虑的。

贪心算法，每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或是最优的。比如在旅行推销员问题中，如果旅行员每次都选择最近的城市，那这就是一种贪心算法。贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是局部最优解能决定全局最优解。简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。

甘特图（Gantt Chart）是的一种条状的图形化概要，最大优点在于易于理解。其中，线条表示在整个期间上计划和实际的活动完成情况。它直观地表明任务计划在什么时候进行，及实际进展与计划要求的对比。

理论上，穷举算法无疑能求解出任何问题的最优解，但是面对典型的 NP-HARD 问题，其时间复杂度是一个天文数字。实际上，我们不可能用穷举算法实现大多数的单机调度问题。

本文我们从穷举算法出发，结合单机调度的性质，适当选择贪心选择条件，以最大程度降低调度总延迟未目标，将贪心选择和工件结合在一起，以获取足够好的解。

§2 模型假设

2.1 背景

1. 每个工件上只可被机器加工一次，且只有在工件到达机器处后才可以被加工。
2. 每个机器某一个时刻最多只能执行一个工件，而且执行过程是非抢占的。

2.2 符号

R_i：　　工件i的释放时间；

S_i：　　工件i的实际开工时间；

P_i：　　工件i的加工时间；

D_i：　　工件i的计划完工时间；

d_i：　　工件i的实际完工时间与计划完工时间之差；

U_i：　　如果工件i在某调度计划中延误，$U_i = 1$，否则$U_i = 0$；

§3 算法设计

3.1 思路

贪心算法并没有固定的算法框架，算法的关键是贪心策略的选择。因此利用贪心算法的这个特性，将单机调度和贪心算法相结合，在整个计算的过程中都以一定的选择策略对工件的优先顺序进行选择。

三种策略悉数如下：

1．优先选择预计交工时间和加工时间乘积最小的

2．优先选择未超时的工件中交工时间最近的，若工件全部存在延迟，则选择交工时间最近的

3．优先选择未超时的工件中交工时间最近的，若工件全部存在延迟，则选择处理时间最短的

由于存在释放时间，所以仅以交工时间最近的作为贪心选择的解不够好，和穷举算法得出的最优解差了 50 以上，经过加工时间的修正之后，可以近似认为这是一个不错的解。

优先选择未超时的工件，这是因为要优先保证未超时的工件不会产生延迟；而工件全部延迟的时候，我们需要可以选择交工时间最近的，也可以选择处理时间最短的，这都是为了在注定延迟的调度中减少延迟时间。

3.2 实现

表 1：本题样例

工件	释放时间	加工时间	交货时间
J1	0	15	21
J2	0	19	25
J3	0	26	30
J4	0	9	13
J5	15	13	28
J6	15	15	40
J7	24	12	65
J8	34	6	75

由于题目中的样例中，工件个数只有 8 个，所以穷举算法也能在很短的时间内给出解。我们利用递归的思想，建立全排列。当排列个数达到 8 个时，根据当前排列顺序，计算出总延迟，同时更新最小总延迟。

01 Scheduling by Exhaustion.cpp

```
void permutation(struct PIECE array[], int head, int tail)
{
    for(int i = head; i <= tail; ++i)
    {
        swap(array[i], array[head]);
        permutation(array, head + 1, tail);
        swap(array[i], array[head]);
    }
}
```

在全排列中，对每一可能的顺序进行操作，先后计算当前时间和延迟时间，并记录当前排序。

01 Scheduling by Exhaustion.cpp

```
for(int i = 0; i <= tail; ++i)
{
    curTime = (curTime >= array[i].release) ? curTime + array[i].process :
array[i].release + array[i].process;
    TimeOfDelay += (curTime > array[i].deliver) ? curTime - array[i].deliver : 0;
    curSequence[i] = array[i].n;
}
```

在穷举算法的基础上没我们很容易就能写出贪心算法，将 permutation 函数改写成 greedyWithPriority 函数。其中，全排列改写成贪心选择。不过，在选择的时候，不能再次选择曾经选择过的工件。因此，我们需要设置一个 mark 数组记录已经选择过的工件。

02 Scheduling by Greedy ver 01.cpp

```
int greedyWithPriority(struct PIECE array[])

for(int i = 0; i < numOfPiece; ++i)
{
```

```
    if(mark[i] == 0)
    {
        curPriority = array[i].deliver*array[i].process;
        if(maxPriority > curPriority)
        {
            maxPriority = curPriority;
            indexOfMaxPriority = i;
        }
    }
}
mark[indexOfMaxPriority] = 1;
sequence[cnt++] = array[indexOfMaxPriority].n;
```

在贪心算法一中，我们只需设置一个 `priority` 并实时更新它即可。但在贪心算法三中，我们需要建立已超时序列 `timeout` 和未超时序列 `notimeout`，并时刻维护。

`03 Scheduling by Greedy ver 02.cpp`

```
if(array[i].deliver - curTime <= 0)
    timeout[indexOfTimeout++] = i;
else
    notimeout[indexOfNoTimeout++] = i;

qsort(notimeout, indexOfNoTimeout, sizeof(notimeout[0]), cmp);
qsort(timeout, indexOfTimeout, sizeof(timeout[0]), cmp);
indexOfMaxPriority = (indexOfNoTimeout > 0) ? notimeout[0] : timeout[0];
```

```
int cmp(const void *a, const void *b)//用于未超时序列的比较函数
{
    return piece[*(int *)a].deliver - piece[*(int *)b].deliver;
}
```

`04 Scheduling by Greedy ver 03.cpp`

```
if(array[i].deliver - curTime <= 0)
    timeout[indexOfTimeout++] = i;
else
    notimeout[indexOfNoTimeout++] = i;

qsort(notimeout, indexOfNoTimeout, sizeof(notimeout[0]), cmp1);
qsort(timeout, indexOfTimeout, sizeof(timeout[0]), cmp2);
```

```
indexOfMaxPriority = (indexOfNoTimeout > 0) ? notimeout[0] : timeout[0];
```

```
int cmp1(const void *a, const void *b)//用于未超时序列的比较函数
{
    return piece[*(int *)a].deliver - piece[*(int *)b].deliver;
}
```

```
int cmp2(const void *a, const void *b)//用于已超时序列的比较函数
{
    return piece[*(int *)a].process - piece[*(int *)b].process;
}
```

§4 编程实验

4.1 前期设置

贪心算法代码简单，很容易实现，本文选择 C++作为编程语言，CodeBlocks、Visual Studio 和 Visual Studio Code 作为编程环境。

由于贪心算具有贪心选择性质，仅在线性时间内即可给出接近于最优的解，所以运行时间可以忽略。

硬件方面，本人使用的电脑型号为 Lenovo Yoga 710-14，Windows10 2018 春季创意者版，超极本类别，6 代中央处理器。

4.2 结果

首先我们运行编写好的程序，得到如下结果。其中中文数字表示工件排在调度序列的第几位。

表 2：贪心算法与穷举结果的对比

	穷举	贪心一	贪心二	贪心三
解	173	209	212	201
一	4	4	4	4
二	1	1	1	1
三	5	5	2	2

四	6	8	7	7
五	7	2	8	8
六	8	6	5	5
七	2	3	3	6
八	3	7	6	3

根据已经得到的四种调度，我们可以制作如下单机调度的甘特图。其中，纵坐标代表工件对应的序号，横坐标代表工件的加工时间。

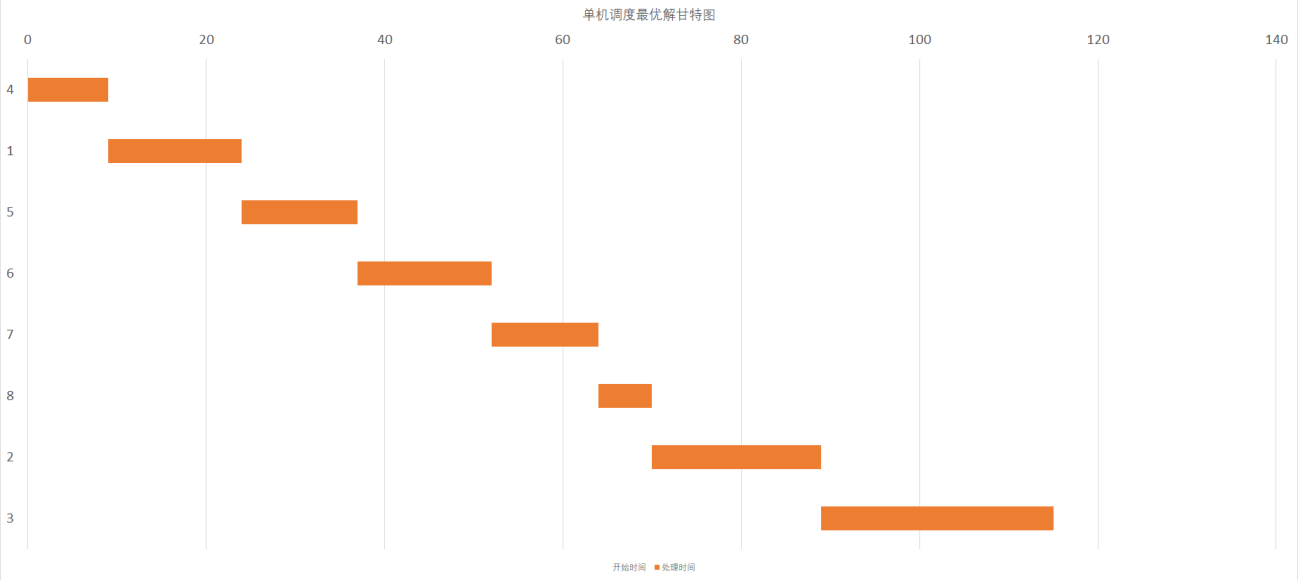

图 1：穷举算法得出的最优调度

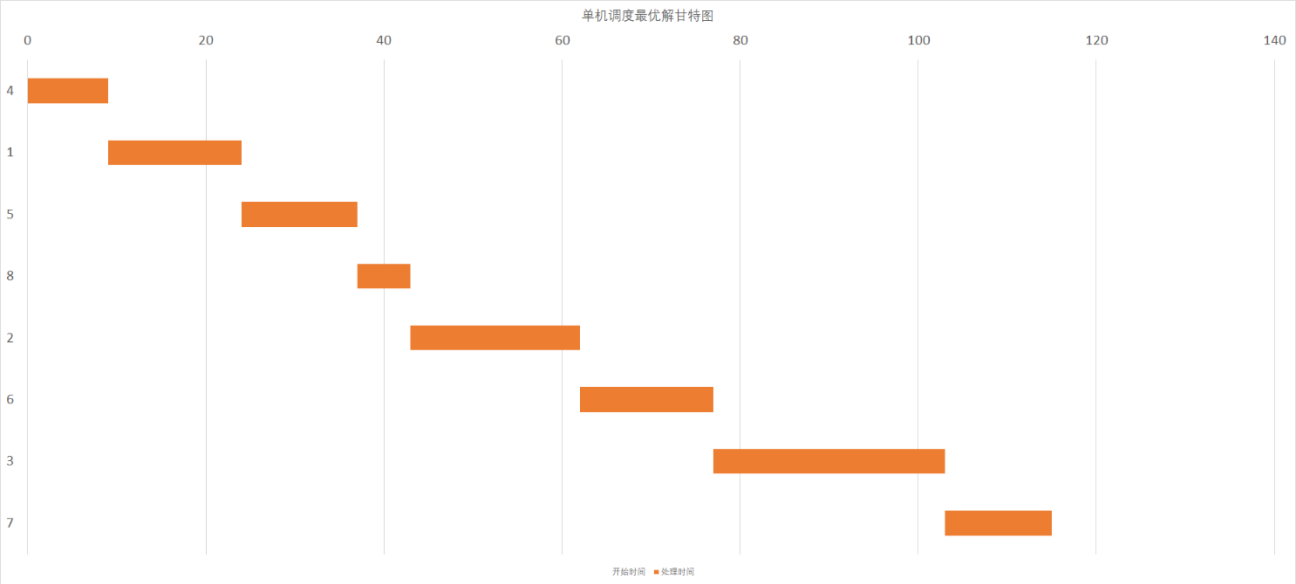

图 2：贪心算法一得出的调度

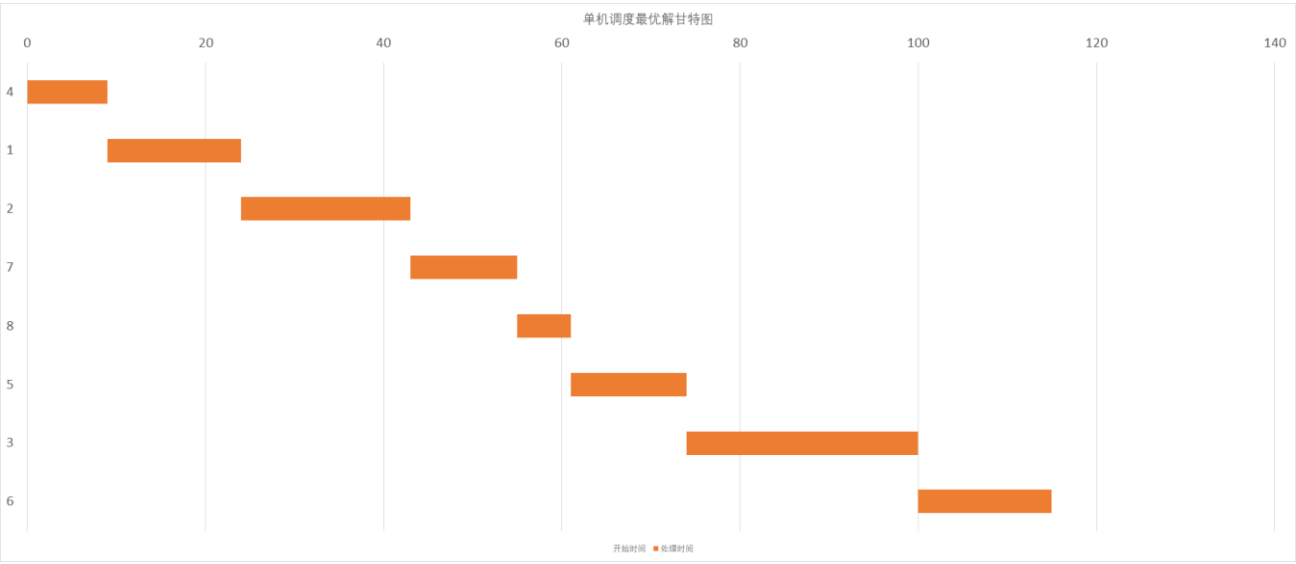

图 3：贪心算法二得出的调度

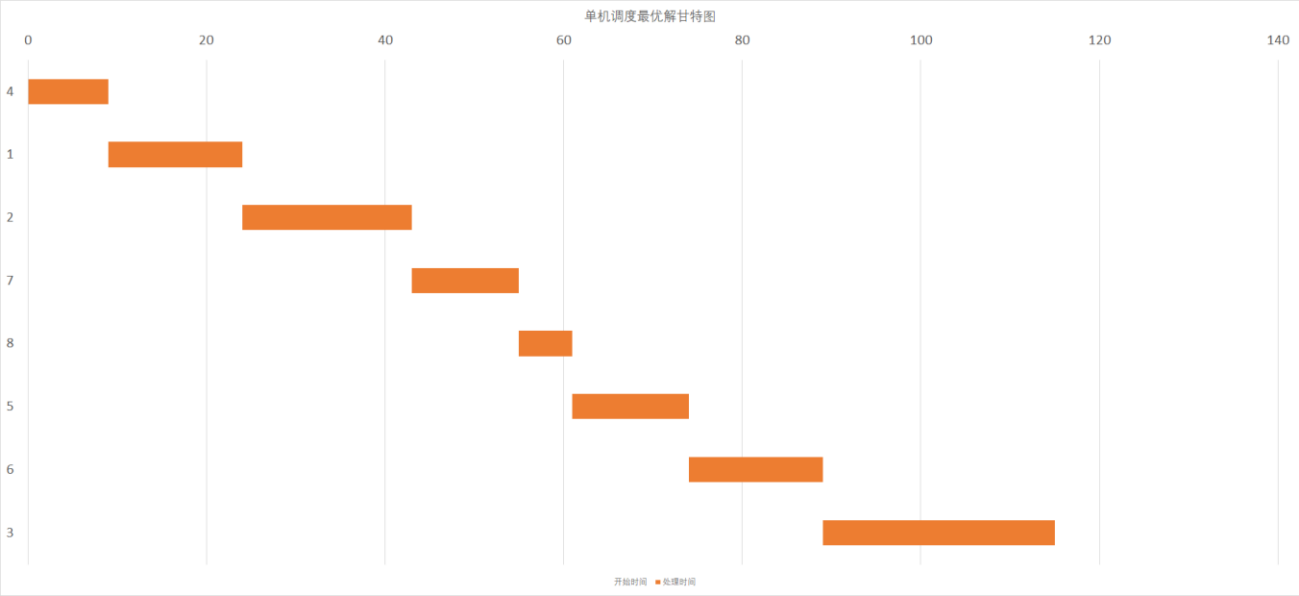

图 4：贪心算法三得出的调度

我们可以清楚的看到，虽然贪心算法可以很快地给出接近最优的解，但是实验结果表明，与贪心算法相比，穷举算法的求解质量好于本文给出的三种贪心算法。但是，当解共建成阶乘速率增长时，贪心算法的效率还是很客观的。

§5 总结评价

当用穷举算法去搜索在最优解 173+50 以内的解时，我们发现在 8!=40320 个可能解中，至少有几十个接近最优解的解，所以可能存在贪心算法使得贪心选择的解和最优解相差无几，而这是我们需要去寻找的。

单机调度既简单又复杂，简单在易于理解，难在无法在多项式时间内求得最优解。同时单机调度又有很多的形式，比如在没有准备时间的情况下，有着调度中最大交货延迟不小于加工时间之和这一特殊的性质。所以，在简单的问题中，还有许许多多的问题等待着我们去解决。

§6 参考文献

[1] http://acm.sgu.ru/problem.php?contest=0&problem=259