运用模拟退火算法解决调度问题

董若扬　学号：1120202944

摘　要: 流水车间调度问题可以描述为 n 个工件要在 m 台机器上加工，每个工件需要经过 m 道工序，每道工序要求不同的机器加工，每个机器同时只能加工一个工件，一个工件不能同时在不同的机器上加工。工件在机器上的加工时间是给定的。问题的目标是求 n 个工件的加工顺序，使总完工时间达到最小。本文采用登山算法与模拟退火算法解决此调度问题，比较这两种局部搜索算法的区别与解的质量，并探究解决该问题合适的登山参数与退火参数。实验结果表明在小规模的问题中登山算法表现出较好的性能，但随着问题规模的扩大，登山算法不能保证每次都能得出可接受的结果，当问题规模较大时，登山算法不能在可接受的尝试次数下得到可接受的结果，陷入局部最优。而模拟退火算法性能表现平稳，无论问题规模大小，都能在较短的时间内得到可靠的结果，在百倍于样例规模的输入上也能很快得到十分优秀的解。

关键词: 模拟退火算法 登山算法 局部搜索 优化算法 启发式算法

1　引言

在现代的工业企业中，生产环节很多，协作关系复杂，生产连续性强，某一个生产节点的变化，如某一机器发生故障，某两道工序顺序进行了调整等等，往往会波及整个生产系统的运行。一般在工业企业中，需要一个专门的部门去组织实现这项任务。

流水车间（Flow Shop ）调度问题是很多实际流水线生产调度问题的简化模型，其研究具有重要的理论意义和工程价值，也是目前研究广泛的一类典型调度问题。

流水车间调度问题可以描述为：n 个工件要在 m 台机器上加工，每个工件需要经过 m 道工序，每道工序要求不同的机器加工，每个机器同时只能加工一个工件，一个工件不能同时在不同的机器上加工。工件在机器上的加工时间是给定的。问题的目标是求 n 个工件的加工顺序，使总完工时间达到最小。

本文将采用登山算法和模拟退火算法解决此调度问题。

登山算法随机选择一个起点。每次拿相邻点与当前点进行比对,取两者中较优者,作为下一步，重复操作直至该点的邻近点中不再有比其好的点。选择该点作为本次爬山的顶点，即为该算法获得的最优解。

模拟退火算法在登山算法的基础上模拟了物理上的退火过程，从某一较高初温出发，伴随温度参数的不断下降，结合一定的概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。

无论是登山算法还是模拟退火算法，都是一种局部搜索启发式算法，其本意就是通过很小的计算量达到近似遍历搜索的结果，于是需要在运行时间与计算结果准确度中作出平衡，否则优化算法退化成随机搜索不能体现其优势。无论对参数如何控制，计算量增加并不能确保优化算法得到最优解。所以为了确保得到最优解，需要进行多次实验。为了提高效率，节约时间，本文中采用多线程并发式求解，根据不同计算机处理器核心可用线程数的不同，自动选择合适的并发线程，最大限度利用计算资源。以本人 16 线程处理器为例，结果表明计算速度

较单线程串行式运行提升了 8 倍。本文在提供的用例测试的输入规模下表现良好，收敛速度快，准确性高，平均每次仅耗时 0.1 秒，有接近于 1 的概率得到全局最优解。

本文后续部分组织如下。第 2 节详细陈述使用的方法，第 3 节报告实验结果，第 4 节对讨论本文的方法并总结全文。

2 算法设计

2.1 程序总体框架设计

（1）预设初始温度 1500℃，降温系数 0.99，临界温度 1e-4℃，同一温度下重复次数 206 次。

（2）输入工件数，机器数，作为二维数组的行和列的长度。输入各工件在各工件上的加工时间，将时间储存在二维数组 data 中。并在填满二维数组 data 后检查是否还有待输入的数据，若有，提醒操作者检查输入是否规范。

以多线程并发式执行以下过程：

（3）将 data 中数据复制到当前子线程下的二维数组 current，next。

开始模拟退火：

（4）运用动态规划求“最大可行折线和”的方法求任一顺序下工件的最小加工时间。

（5）当温度大于临界温度时执行循环，每次循环依次进行：1.任意交换两个工件的加工顺序。2.运用算法（Ⅰ）计算交换后的最小加工时间。3.处理：若交换后加工时间变短，则保留这次交换。若交换后加工时间变长，则有 exp(-delta / t)的概率保留此次交换，否则恢复交换前的顺序。（delta 为交换后的最小时间-交换前的最小时间，此处 delta 恒为正数，t 为当前温度）4.降温，当前温度乘以降温系数得到下一轮的温度。当温度低于临界温度时，跳出循环。

（若采用登山算法：

（5）重复进行 N 次登山操作，每次操作执行：1..任意交换两个工件的加工顺序。2.计算交换后的最小加工时间。3.处理：若交换后加工时间变短，则保留这次交换，否则恢复交换前的顺序。

）

（6）将该线程得到的结果保存在结果列表，并向控制台输出结果。

结束多线程。

（7）将结果列表中所有线程计算得到的结果按升序排序，计算程序总运行时间，然后读入文件。

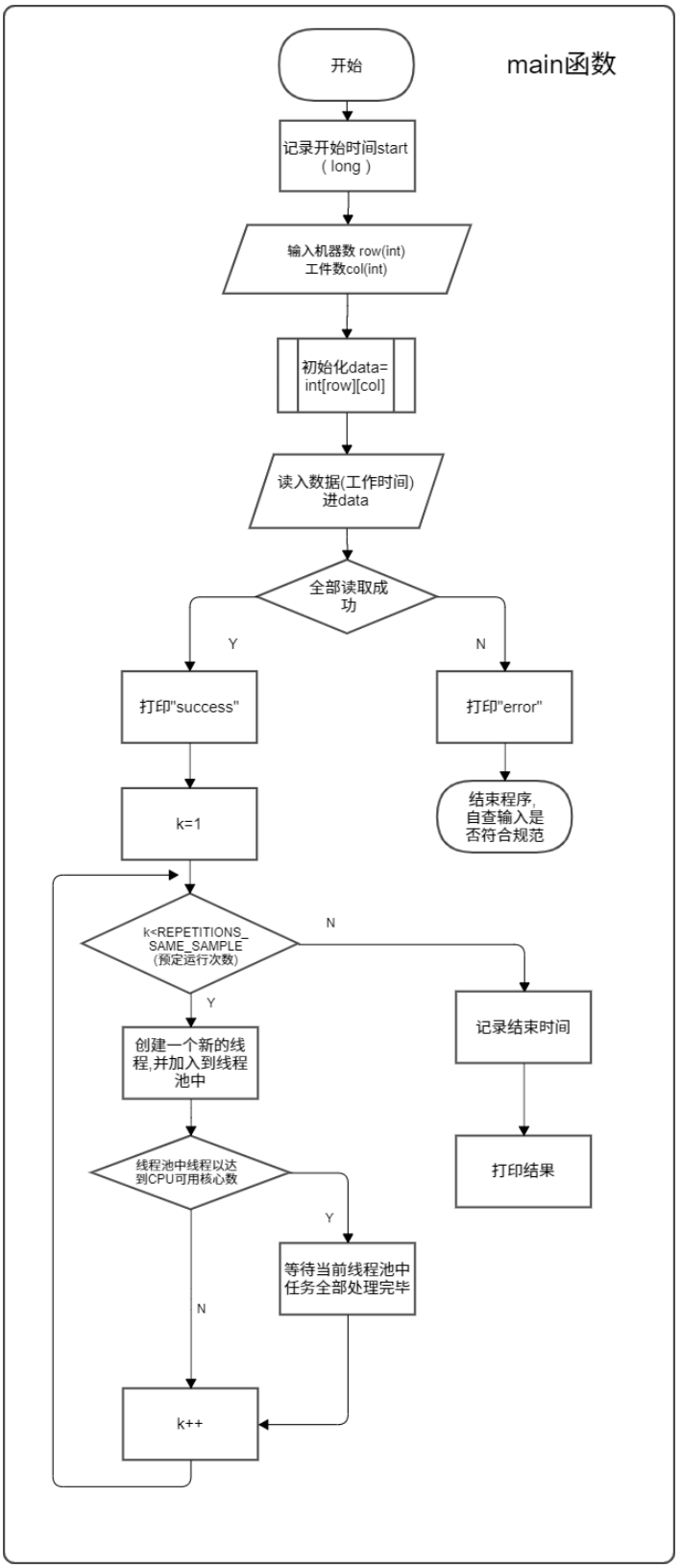

开始
main函数
记录开始时间start
(long)
输入机器数 row(int)
工件数col(int)
初始化data=
int[row][col]
读入数据(工作时间)
进data
全部读取成
功
Y
N
打印"success"
打印"error"
结束程序,
自查输入是
否符合规范
k=1
k<REPETITIONS_
SAME_SAMPLE
(预定运行次数)
N
Y
记录结束时间
创建一个新的线
程,并加入到线程
池中
打印结果
线程池中线程以达
到CPU可用核心数
Y
等待当前线程池中
任务全部处理完毕
N
k++

图 1 程序主体流程图

2.2 算法介绍

2.2.1 计算任一顺序下的最小加工时间：

先介绍一个引理:

设有 m 台机床 M_1，M_2，……，M_m 和 n 个零件 J_1，J_2，……，J_n。记零件 J_i 在机床 M_k 上的加工时间为 a_{ki}，当零件的加工顺序 ω=（ω_1，……，ω_n）给定时，其中（ω_1，……，ω_n）表示（1，……，n）的一种排列，我们可将 a_{ki} 排成矩阵

$$A(\omega)=\begin{bmatrix} a_{1\omega 1} & \cdots & a_{1\omega n} \\ \vdots & \ddots & \vdots \\ a_{m\omega 1} & \cdots & a_{m\omega n} \end{bmatrix}$$

下面给出关于 A (ω)的可行线与可行和的概念：

定义 1. 将矩阵 A（ω）中 $a_{1\omega 1}$ 与 $a_{m\omega n}$ 用一条折线连接起来，此折线只能向右水平延伸或向下竖直延伸，其顶点只能是 a_{ki}，这种折线称之为对应于 ω 的一可行线，全体可行线所成集合为 L（ω）。

定义 2. 设 l（ω）为一可行线。则称下式为对应于 l（ω）的可行和

$$\sum_{(k,\ i)\in l(\omega)} a_{ki}$$

引理. 设 ω=（ω_1，……，ω_n）为一加工顺序，如果每个零件都须依次通过 M_1，M_2，……，M_m，且零件在各机床上加工顺序一致，则从 M_1 开始加工零件 J_{m1} 到 M_m 加工完零件 J_{mn} 为止， 这段时间之和 T（ω）的最小值，等于矩阵 A（ω）的所有可行和中的最大值。[1]

算法：设 dp(i, j)为从用时矩阵左上角走至单元格 (i ,j)的用时最大累计值，易得到以下递推关系：dp(i,j)等于 dp(i,j-1)和 dp(i-1,j)中的较大值加上当前单元格用时 value(i,j)。dp(i,j)=max[dp(i,j−1),dp(i−1,j)]+value(i,j)。因此，可用动态规划解决此问题，以上公式便为转移方程。

状态定义：设动态规划矩阵 dp，*dp*(*i*,*j*) 代表从矩阵的左上角开始到达单元格 (*i*,*j*) 时加工用时的最大累计值。

转移方程：

当 i = 0 且 j = 0 时，为起始元素；*dp*(*i*,*j*)= *value*(*i*,*j*)

当 i = 0 且 j≠0 时，为矩阵第一行元素，只可从左边到达；*dp*(*i*,*j*)= *value* (*i*,*j*)+*dp*(*i*,*j*−1)

当 i ≠0 且 j=0 时，为矩阵第一列元素，只可从上边到达；*dp*(*i*,*j*)= *value* (*i*,*j*)+*dp*(*i*−1,*j*)

当 i ≠0 且 j≠0 时，可从左边或上边到达；*dp*(*i*,*j*)= *value* (*i*,*j*)+max[*dp*(*i*−1,*j*),*dp*(*i*,*j*−1)]

初始状态： dp[0][0] = value[0][0]

返回值： dp[m-1][n-1] ，m, n 分别为矩阵的行高和列宽，即返回 *dp* 矩阵右下角元素。

复杂度：设机器数为 M，工件数为 N，动态规划需遍历整个 data 矩阵，使用 O(MN)时间。需要额外 M 行 N 列的 dp 矩阵储存中间值，使用额外 O(MN)空间。若直接在 data 矩阵上操作，可以将空间复杂度降至 O(1)，但是原先 data 矩阵储存了工件加工的顺序信息，如果要保留信息，进行恢复操作的话时间复杂度要乘以常数项。由于制约该算法性能的主要因素是运行时间，所以采用空间换时间的策略。另外当 data 矩阵很大时， i = 0 或 j = 0 的情况仅占极少数，其余循环每轮都冗余了一次判断。因此，可先初始化矩阵第一行和第一列，再开始遍历递推。这样可以减小时间复杂度中的系数，但仍使用 O(MN)时间。

伪代码：

```
for (int i = 1; i < 工件数; i++)
    for (int j = 1; j < 机器数; j++)
        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + value[i][j];
```

流程图：

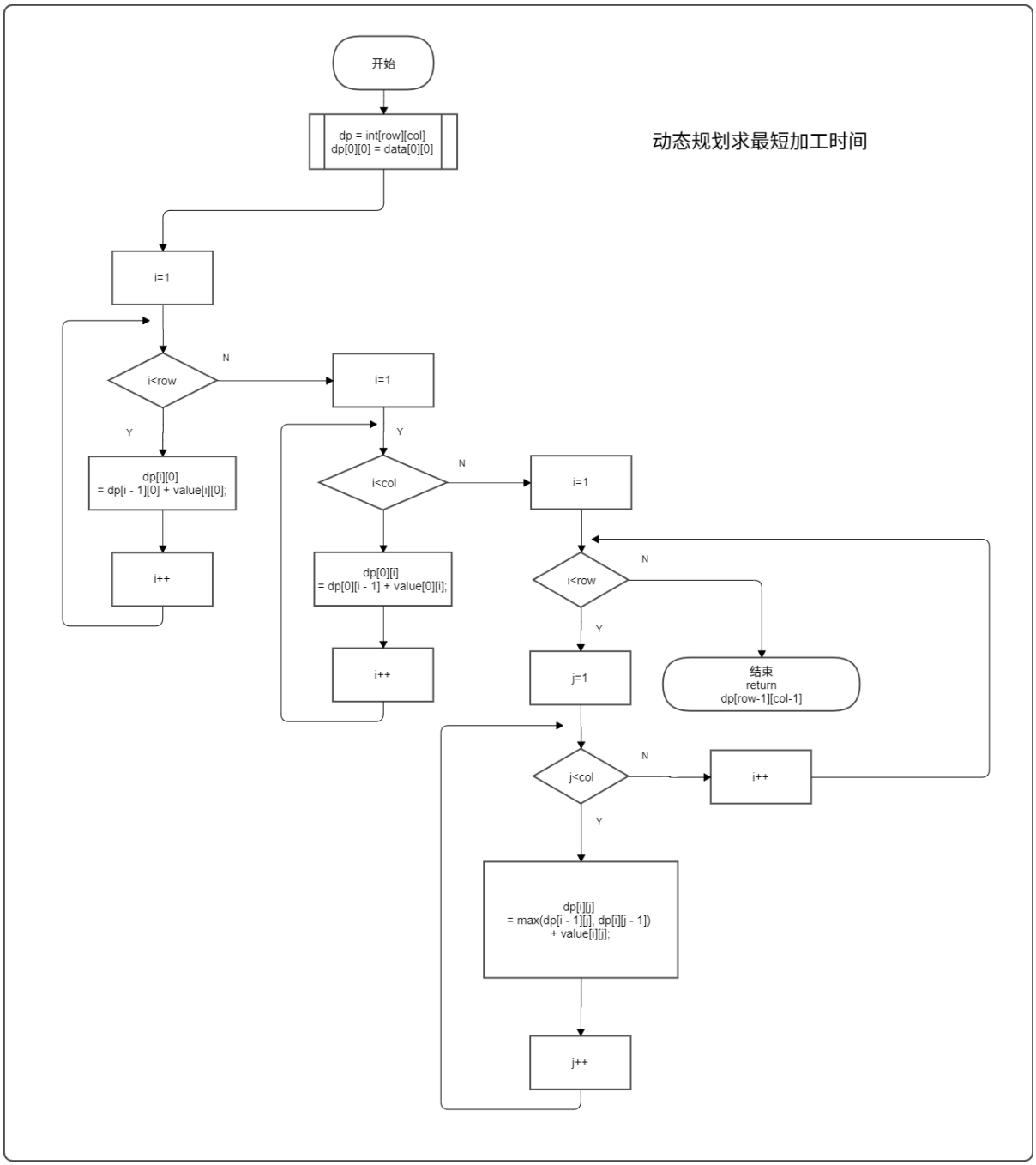


图 2 动态规划求解流程图

2.2.2　登山算法

算法：重复进行 D 次登山操作，每次操作执行：1. .任意交换两个工件的加工顺序。2.计算交换后的最小加工时间。3.处理：若交换后加工时间变短，则保留这次交换，否则恢复交换前的顺序。

复杂度：设登山次数为 D，则时间复杂度为 O(D)，考虑到计算耗时的时间复杂度为 O（MN），其中 M 为机器数量，N 为工件数量，交换零件加工顺序需要交换矩阵的两行，时间复杂度是 O（M），最终时间复杂度为 O（D *（M+MN+M））, 因为实际问题中 N≥2，所以总时间复杂度为 O (DMN)

伪代码：

```
for (int i = 0; i < 登山次数; i++) {
    随机交换两个工件的加工顺序;
    计算交换后的耗时;
    if (交换后耗时更短) {
        更新答案;
        保存本次矩阵更改;
    } else 还原本次矩阵更改;
}
```

流程图：

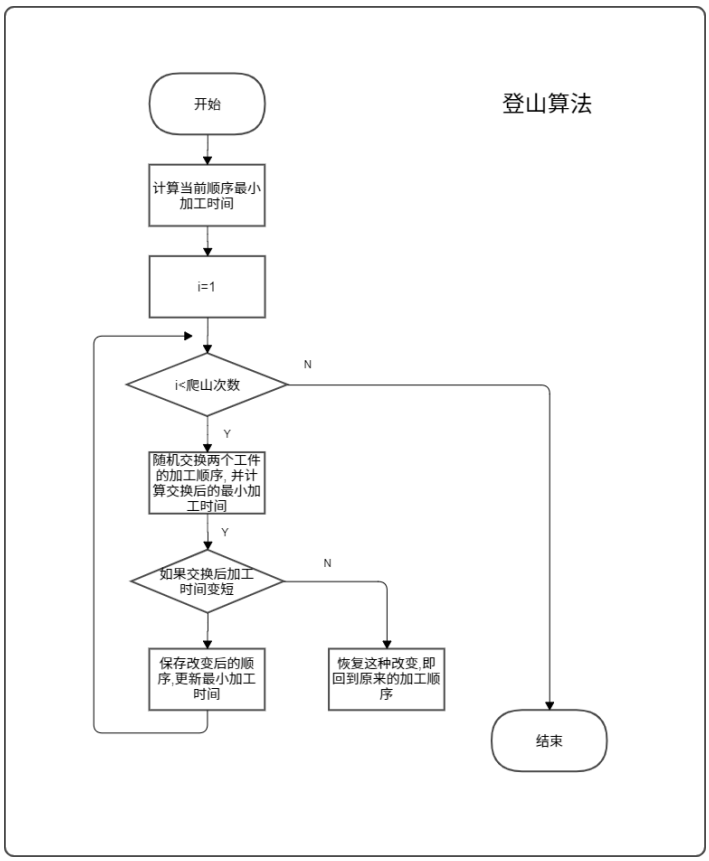


图 3 登山算法伪代码

2.2.3 模拟退火算法

算法：运用动态规划求“最大可行折线和”的方法求任一顺序下工件的最小加工时间。当温度大于临界温度时执行循环，每次循环依次进行：1.任意交换两个工件的加工顺序。2.计算交换后的最小加工时间。3.处理：若交换后加工时间变短，则保留这次交换。若交换后加工时间变长，则有 exp(-delta / t)的概率保留此次交换，否则恢复交换前的顺序。（delta 为交换后的最小时间-交换前的最小时间，此处 delta 恒为正数，t 为当前温度）4.降温，当前温度乘以降温系数得到下一轮的温度。当温度低于临界温度时，跳出循环。

复杂度：设降温次数为 P，重复次数为 Q，则整个退火过程循环使用 O(PQ)时间。考虑到求最小加工时间的时间复杂度为 O（MN），M 为机器数量，N 为工件数量。随机交换两个工件加工顺序的时间复杂度为 O（M）。最终时间复杂度为 O（PQ *（M+MN+M）），模拟退火算法总耗费 O（PQMN）的时间。

伪代码：

```
while (t > 临界温度) {
    for (int num = 0; num < 重复次数; num++) {
        随机交换两个工件的加工顺序;
        计算交换后的耗时;
        delta = 交换后用时-原用时;
        if (delta < 0) {
            更新答案;
            保存此次矩阵更改;
        } else {
            if (在p=exp(-delta / t)的概率下) {
                更新答案;
                保存此次矩阵更改;
            } else 还原此次矩阵更改;
        }
    }
    t *= 衰退系数;
}
```

流程图：

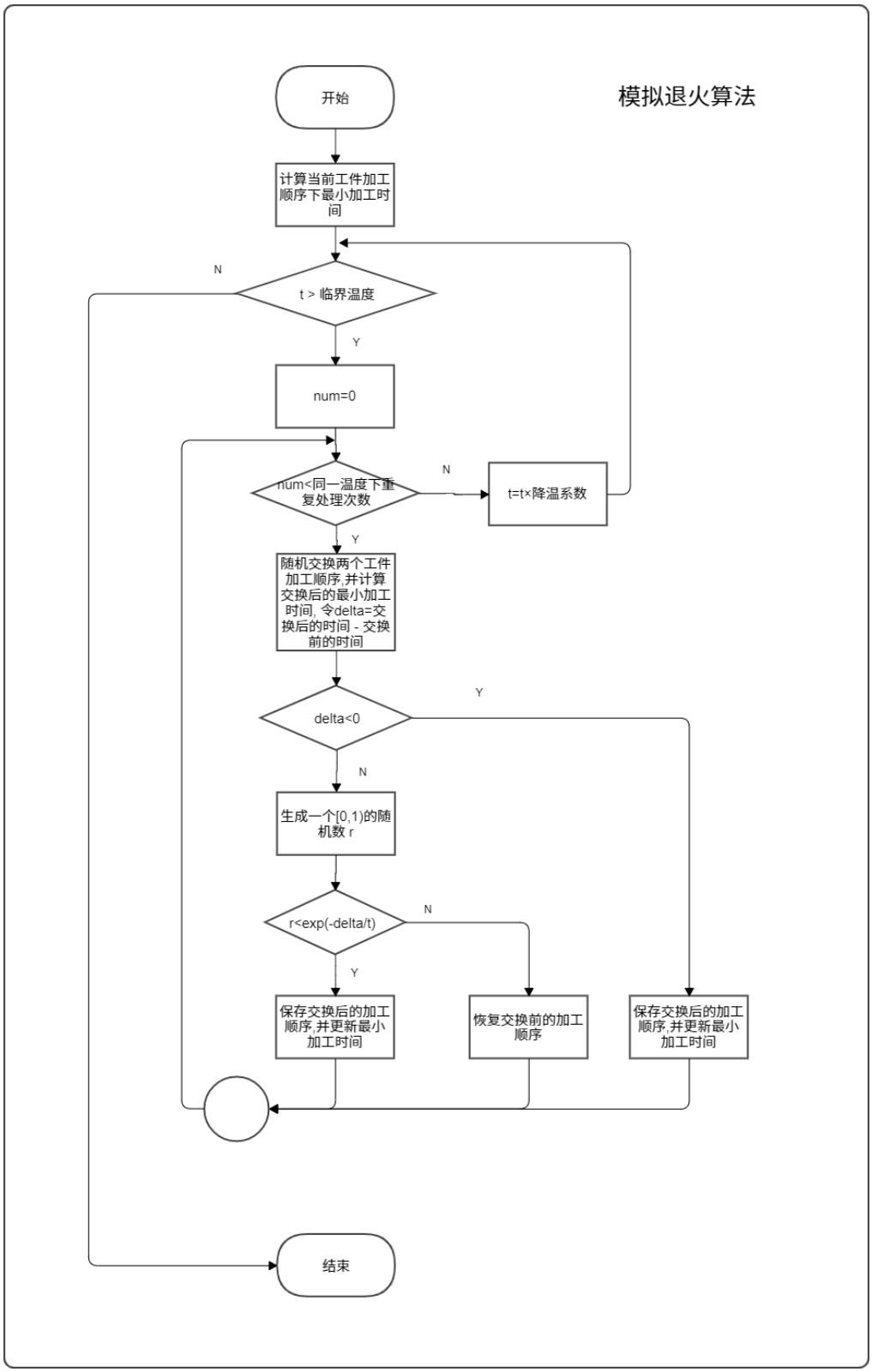


图 4 模拟退火算法伪代码

3 实验

3.1 实验设置

计算机配置：

处理器　AMD Ryzen 7 4800H with Radeon Graphics　2.90 GHz

机带　RAM16.0 GB

系统类型　64 位操作系统, 基于 x64 的处理器

操作系统配置：

版本 Windows 11 家庭中文版

版本 22H2

操作系统版本　22598.200

体验 Windows Feature Experience Pack 1000.22598.200.0

实验环境配置：

Oracle OpenJDK version 17.0.2

运行时间：

在提供的用例的输入规模上，运行时间平均为 100ms

实验参数：

初始温度 1500℃；降温系数：0.99；临界温度 1e-4℃；同一温度下重复次数：206 次；

3.2 实验结果

3.2.1 模拟退火算法在各用例上的表现

表 1 模拟退火算法 128 次运行结果

Instance	SA 最好结果	SA 最差结果	SA 平均结果	极差	平均运行时间/ms
instance 0	7038	7038	7038	0	57
instance 1	8366	8366	8366	0	68
instance 2	7166	7166	7166	0	76
instance 3	7312	7312	7312	0	70
instance 4	8003	8003	8003	0	72
instance 5	7720	7847	7727	127	71
instance 6	1431	1469	1440	38	75
instance 7	1951	1992	1972	41	89
instance 8	1109	1111	1109	2	71
instance 9	1902	1980	1941	78	83
instance 10	3277	3277	3277	0	145

3.2.2 登山算法在各用例上的表现

表 2 登山算法 128 次运行结果

Instance	HC 最好结果	HC 最差结果	HC 平均结果	极差	平均运行时间/ms
instance 0	7038	8130	7100	1092	28
instance 1	8366	9046	8626	680	29
instance 2	7166	8358	7426	1192	30
instance 3	7312	8567	7601	1255	35
instance 4	8003	8611	8264	608	33
instance 5	7720	8483	7880	763	37
instance 6	1451	1626	1517	175	59
instance 7	1964	2150	2040	186	79
instance 8	1111	1187	1136	76	46
instance 9	1943	2108	2012	165	79
instance 10	3284	3482	3358	198	14

3.2.3 模拟退火算法与登山算法 128 次运行结果比较

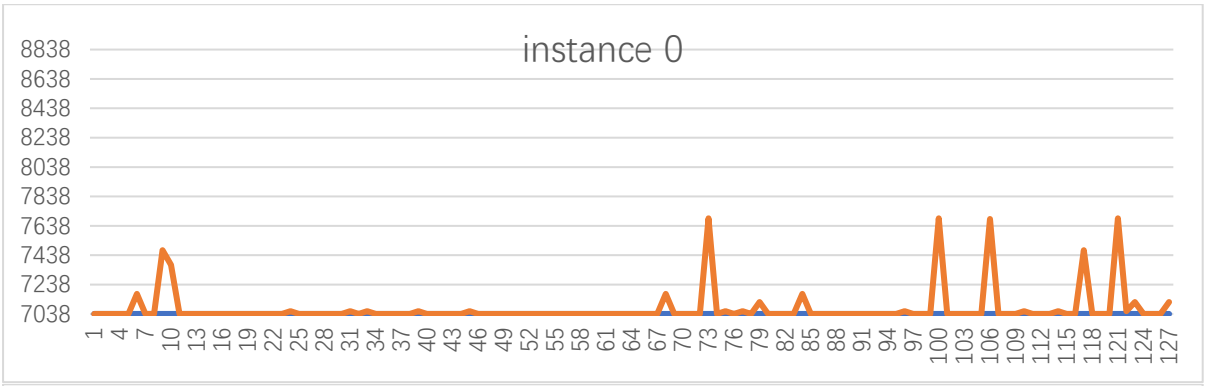
以下图表展示了模拟退火算法（蓝线表示）和登山算法（橙线表示）在 128 次左右的独立运行实验中的表现，其中纵坐标轴的起点值为模拟退火算法得到的最优解值，终点为 128 次随机顺序下加工时间的平均值。这种坐标轴的刻画方式在一定程度上反映了模拟退火算法和登山算法优化能力的强弱。

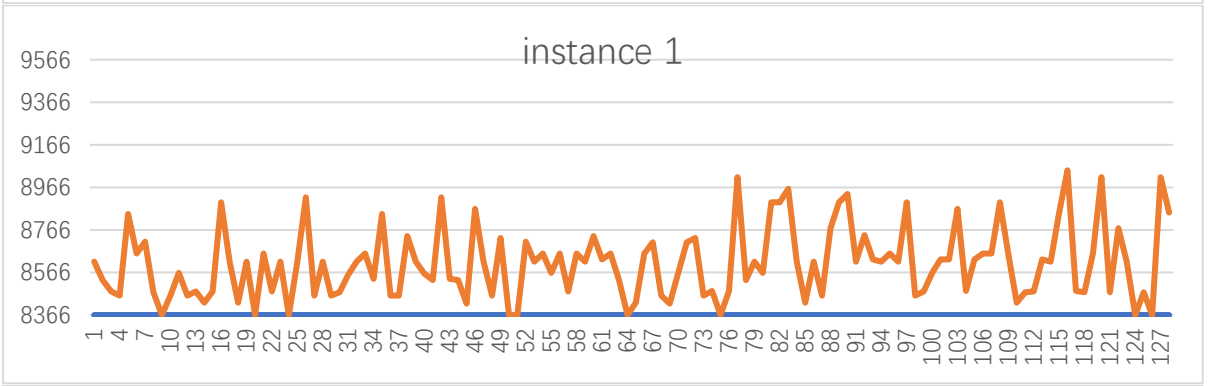
结论：从以下图表可以明显看出以下事实

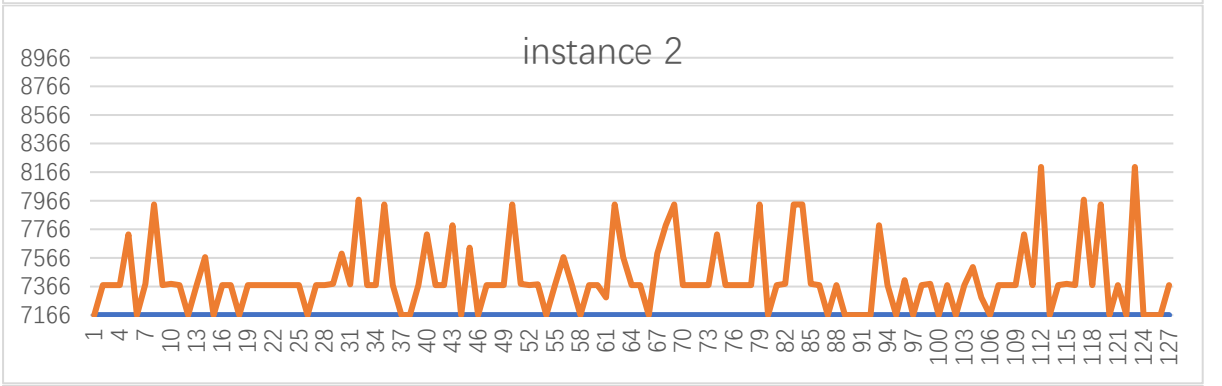
（1） 模拟退火算法的解曲线完全位于登山算法解曲线下方（部分点存在大于登山算法极小值的概率），表明模拟退火算法的优化能力远远强于登山算法。

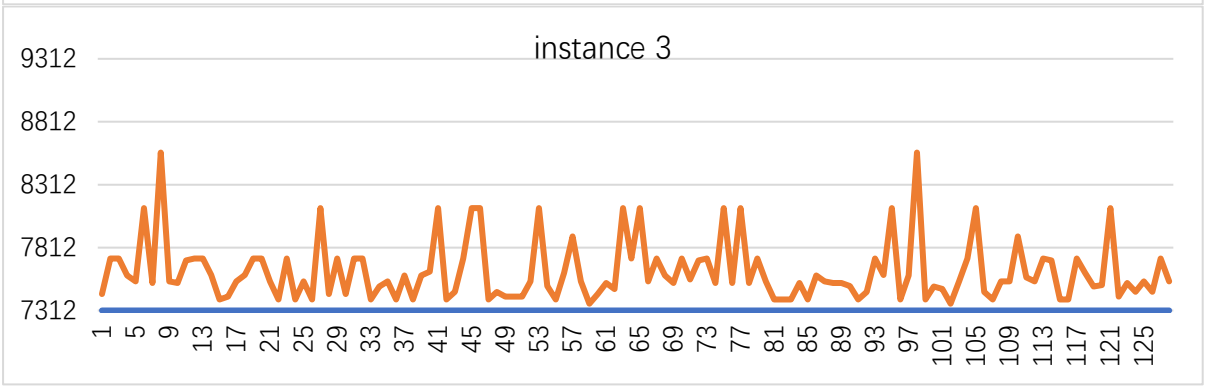
（2） 模拟退火算法计算结果较为稳定，波动程度远远小于登山算法。在问题规模较小时，模拟退火算表现又稳又好，表现为一条直线，而登山算法解曲线偶尔还会有明显高于最优解的“突刺”，表明登山算法很可能仍给出一个很“坏”的结果。而当问题规模提高时，模拟退火算法仍能平稳地发挥，登山算法则出现了大量波动，只有少数点短暂地与最优解相交。但相对于随机取一个顺序得到的结果还是有大幅的提高，能看到登山算法发挥了一定的优化效果。当问题规模比较大时，模拟退火算法的解曲线能保证在大量的实验下，仍可能“探”到最优解。而登山算法则完全位于最优解上方，表现出无论重复多少次实验，都很难足够幸运能在梯度上升方法的指导下避开局部最优解的桎梏的结果。

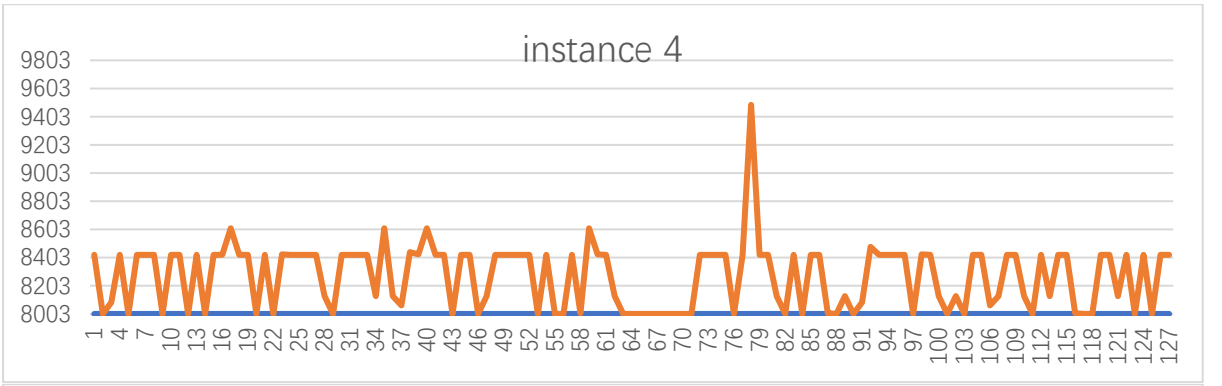
（3） 若只进行一次实验，模拟退火算法在小的问题规模上有大概率得到最优解，在大的问题规模上有大概率能得到质量比较好的解。而登山算法方差太大，发挥极其不稳定，在小规模的问题规模上有一定概率得到最优解，有大概率得到质量比较好的解，但仍存在相当的概率产生非常差的解，在大的问题规模上，登山算法得到一个可接受的质量较好的解几乎不可能，其优化效果十分有限。

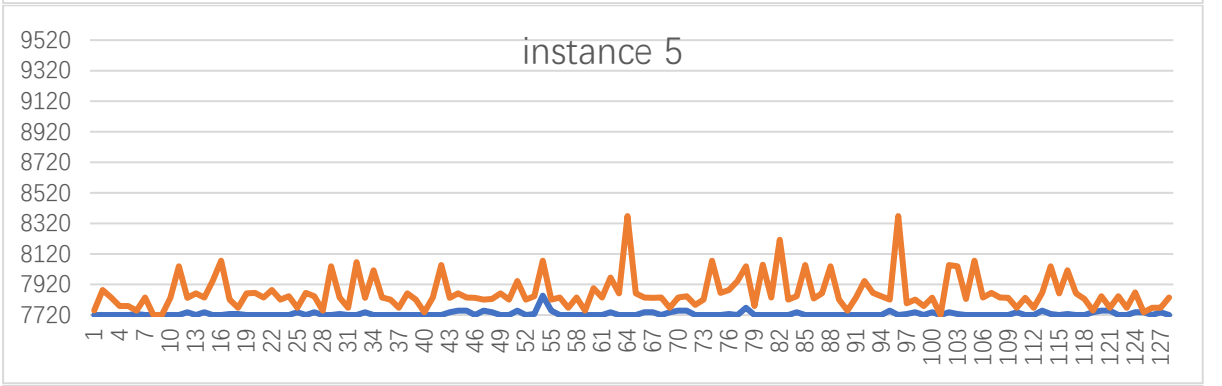

instance 0
8838
8638
8438
8238
8038
7838
7638
7438
7238
7038
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127

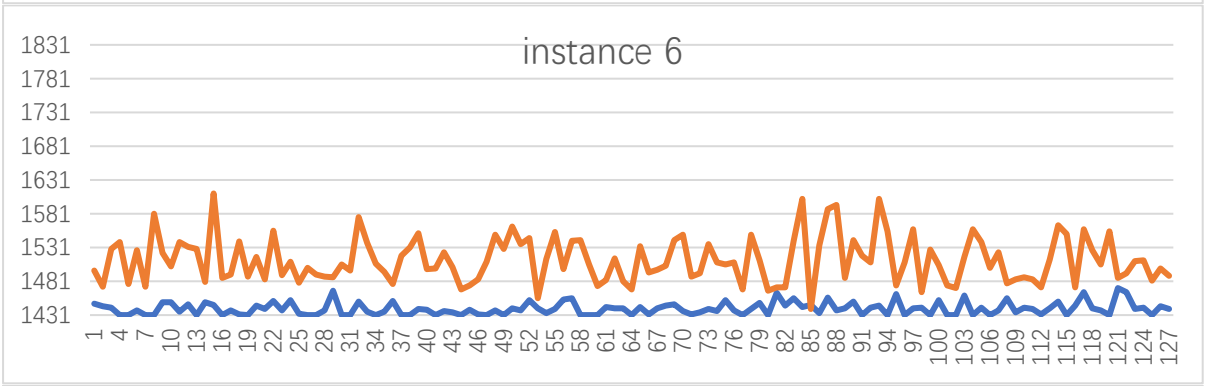

instance 1
9566
9366
9166
8966
8766
8566
8366
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127

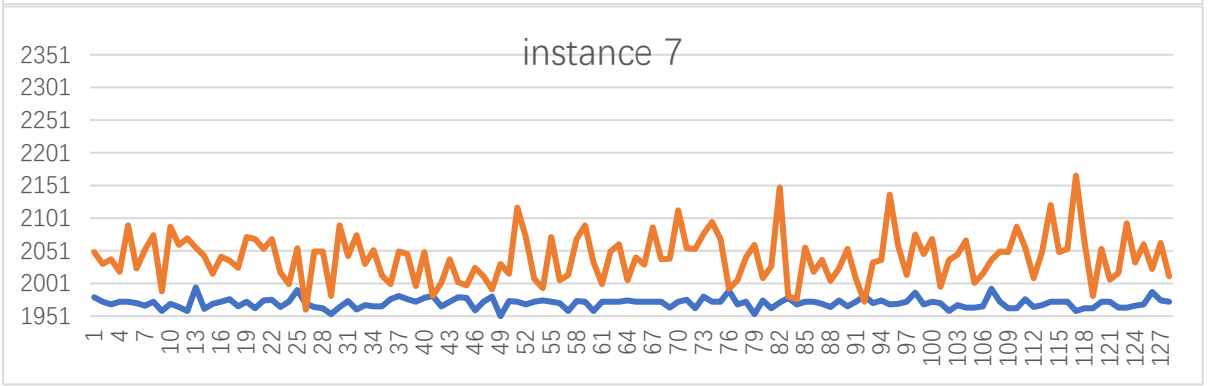

instance 2
8966
8766
8566
8366
8166
7966
7766
7566
7366
7166
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127

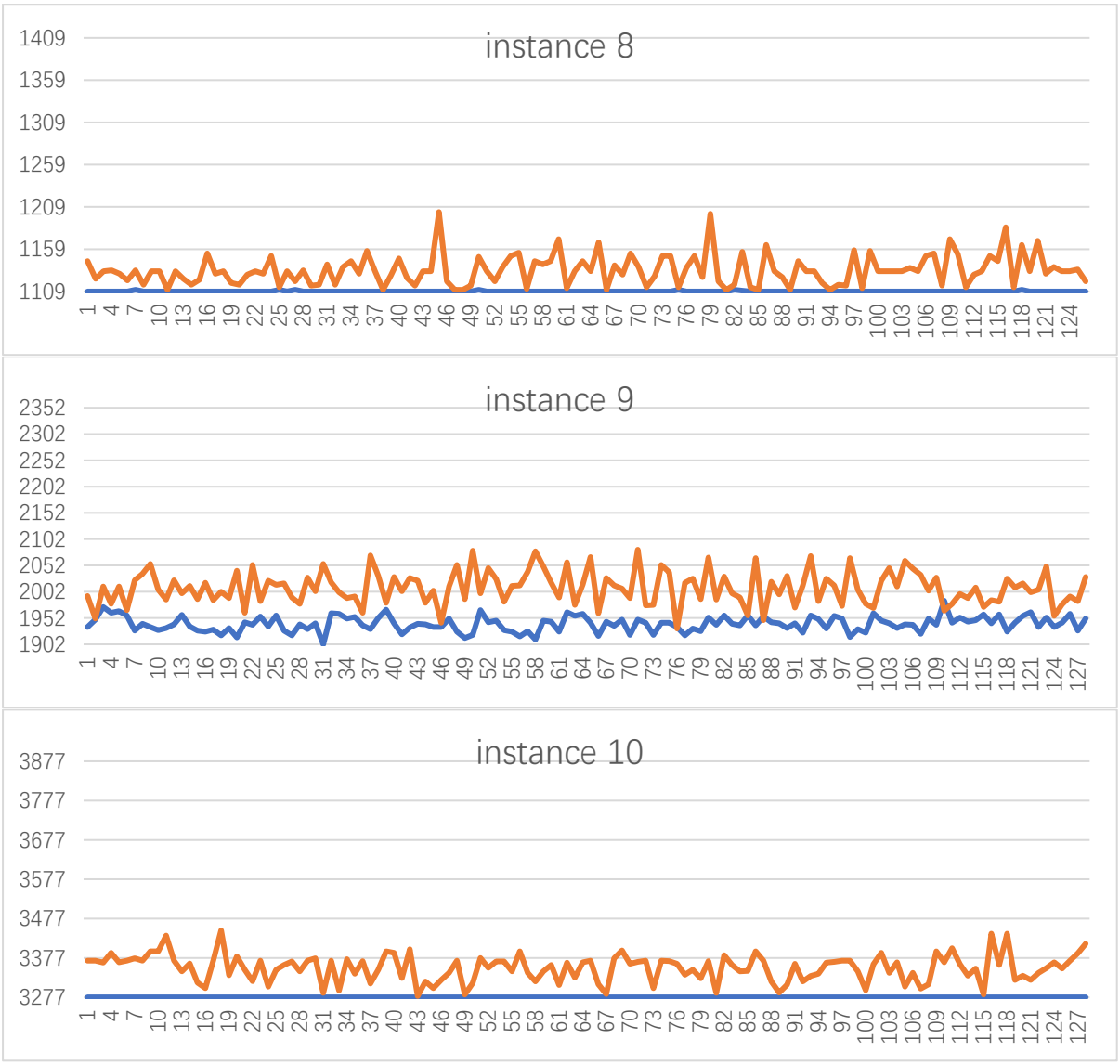

instance 3
9312
8812
8312
7812
7312
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117 121 125


instance 4
9803
9603
9403
9203
9003
8803
8603
8403
8203
8003
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127


instance 5
9520
9320
9120
8920
8720
8520
8320
8120
7920
7720
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127


instance 6
1831
1781
1731
1681
1631
1581
1531
1481
1431
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127


instance 7
2351
2301
2251
2201
2151
2101
2051
2001
1951
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100 103 106 109 112 115 118 121 124 127



3.2.4　模拟退火算法与登山算法 128 次运行结果排序比较

以下图表体现了一组测试样例和十组测试用例在登山算法（橙色线）和模拟退火算法（蓝色线）下解的表现。两个环都是由 128 次独立运行的结果按升序排序后绘制，即从 0 点方向到 12 点方向。

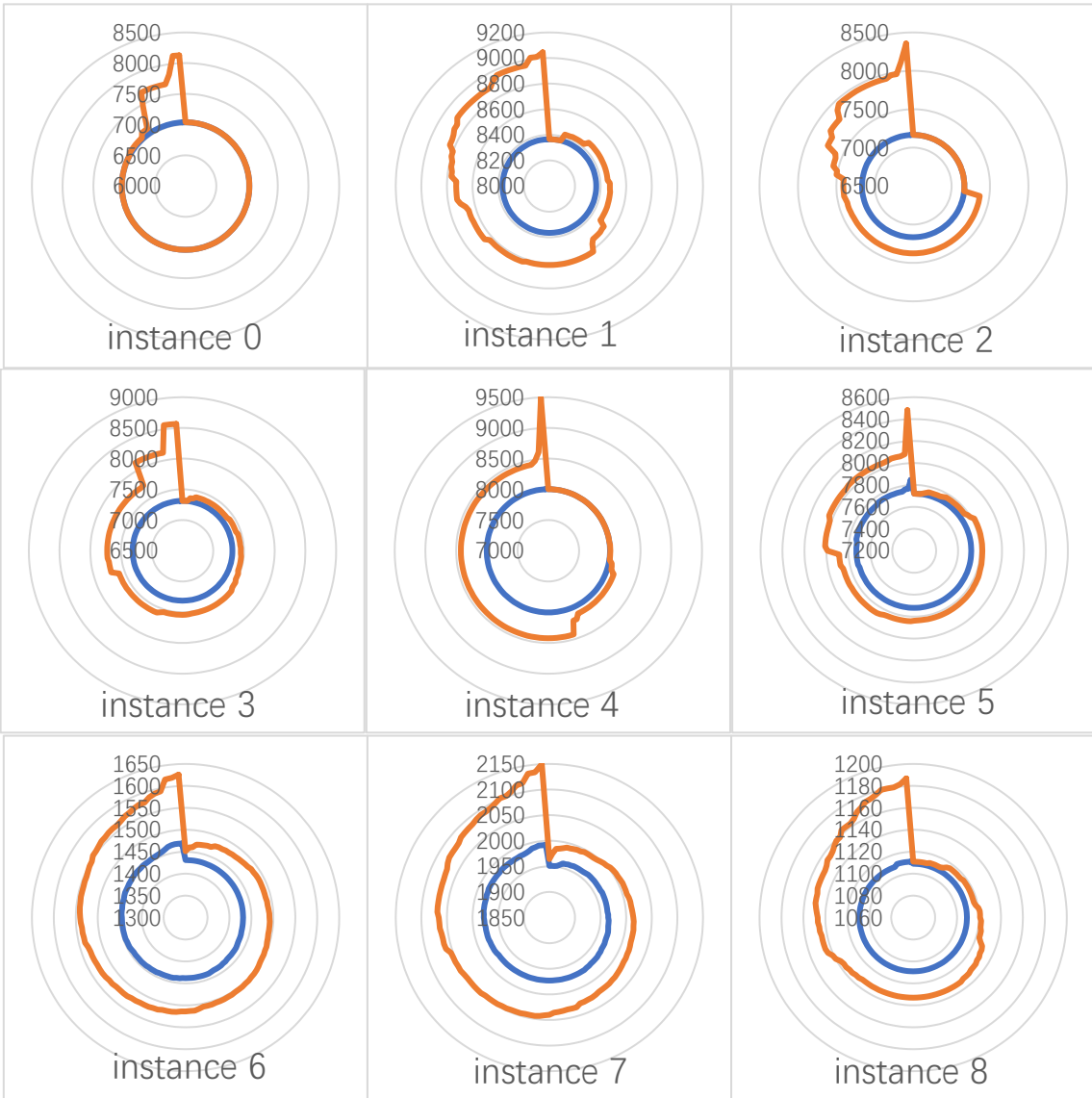
结论：实验数据表明

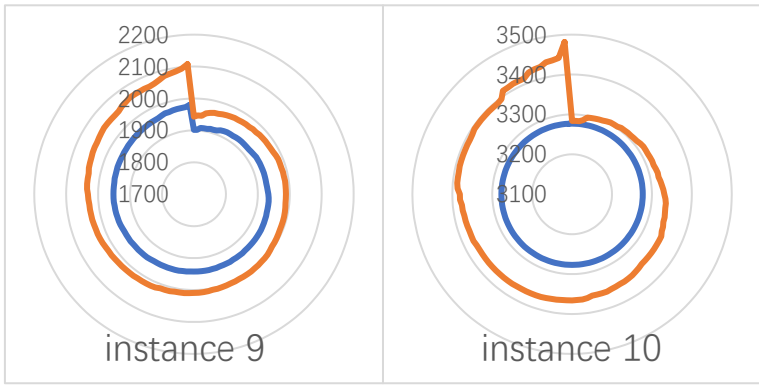
（1）　在问题规模较小时（如 instance 0，1，2，3，4，5），模拟退火算法能在很好的性能表现下以接近于 1 的概率得到最优解，而登山算法通常无法得到最优解，只有在大量重复运行后才能偶尔得到最优解，而且也可能会得到很差的结果。

（2）　随着问题规模的扩大（如 instance 6，8），模拟退火算法仍有极高的概率一次运行就能得到最优解，而登山算法在重复 128 次后也很难得到最优解了，在雷达图中可以看到登山算法的结果和模拟退火算法的结果几乎没有重合了，意味着模拟退火算法的平均结果远远优于登山算法的平均结果。

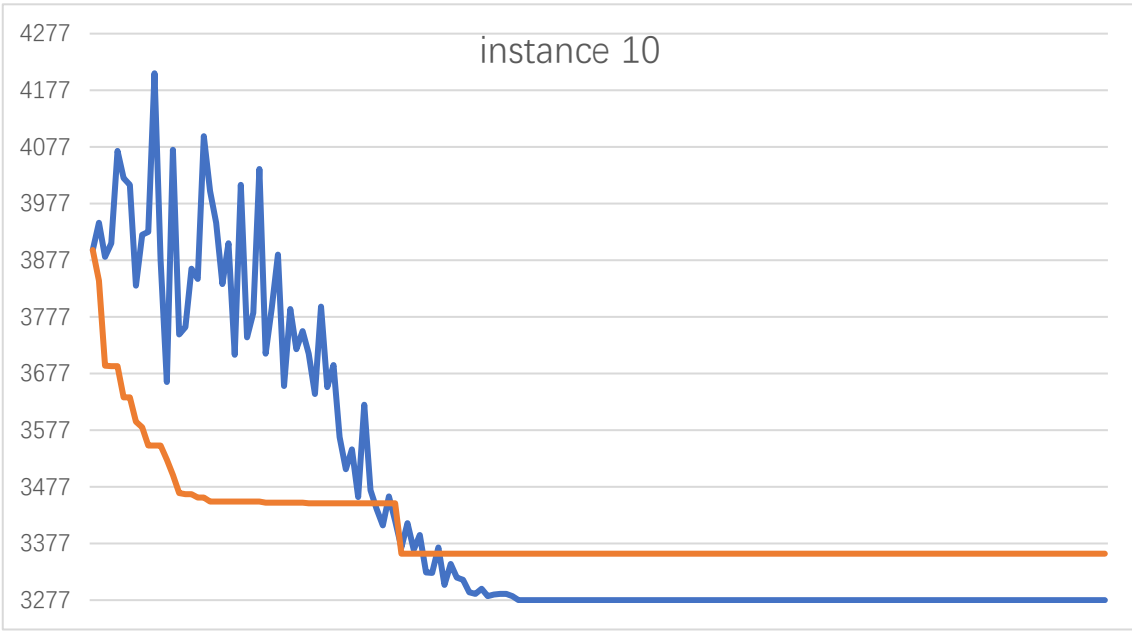
（3） 当问题规模较大时（如 instance 7，9），模拟退火算法虽然不能保证一次运行就得到最优解，但有极高的概率得到可接受的解，而且这个解正常情况下要优于 128 次独立运行的登山算法的最好结果（即任意运行一次模拟退火算法，其结果都是登山算法很难得到的），而且在 128 次独立运行的模拟退火算法的运行结果中，至少有一次找到了最优解。而登山算法在 128 次独立运行下能找到最优解的概率接近于 0，即在可接受的重复次数内登山算法无法找到最优解。

（4） 在特殊的输入条件下（如 instance 10），128 次运行下，模拟退火算法每次都找到了最优解，而登山算法没有一次能找到最优解，且只有很小的概率找到可以接受的解，解的平均质量远远低于模拟退火算法。





3.2.5　用例 10 上登山算法与模拟退火算法的优化过程

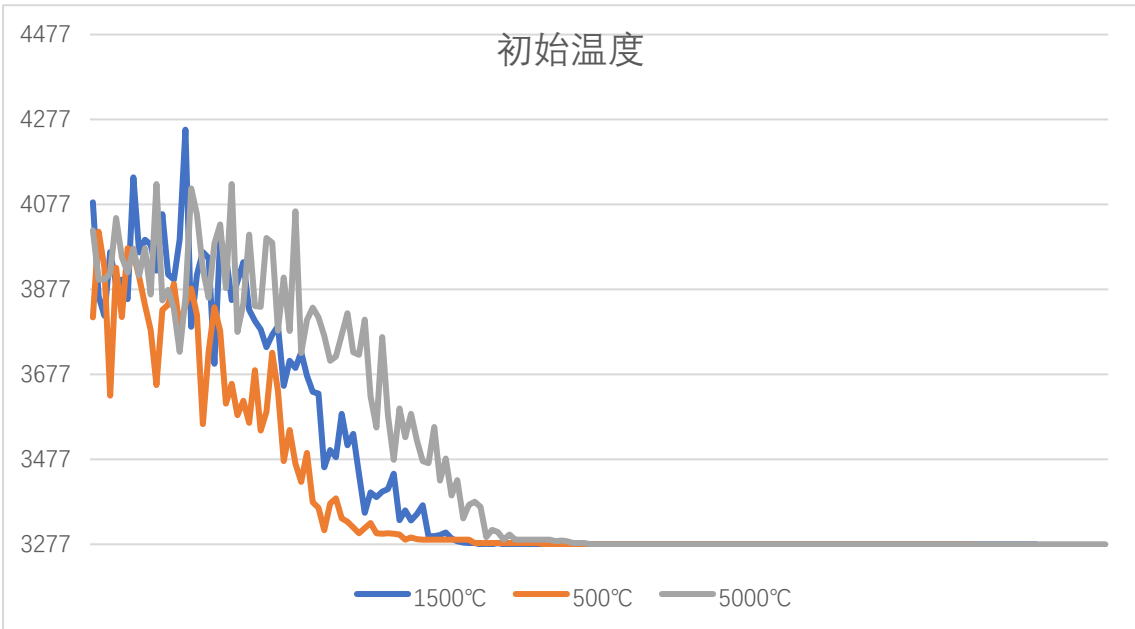


登山算法与模拟退火算法在用例 10 上的优化过程

分析：

登山算法很快就开始下降，而模拟退火算法在运行一段时间后还出现了差于初始顺序的结果，随后一段时间模拟退火算法的结果一直在震荡，但从震荡的峰值来看其趋势是在减小的，在运行还不到一半的时候其结果就优于登山算法，并很快，在程序运行到一半时就找到了最优解。登山算法在起初显著地下降后，区域稳定一段时间后发现改进的顺序后变无法再通过仅交换两个工件的加工顺序来减小加工时间了，陷入了局部最优。

3.2.6 初始温度对模拟退火算法的影响



初始温度对模拟退火算法的影响

模拟退火算法最重要的判断条件

exp（delta/t）>random.nextDouble() (0~1 的随机数)

因为 0-1 之间随机数期望是 0.5

Ln(0.5)= -0.69

也就是说-0.69 < delta/t <0 时, exp（delta/t）>0.5

所以大致可以将模拟退火算法理解成只要差值的绝对值在 t 的 0.7 倍以内就会被采用。因为温度 t 随着迭代逐渐降低，t*0.7 逐渐变得更小，实现了退火。

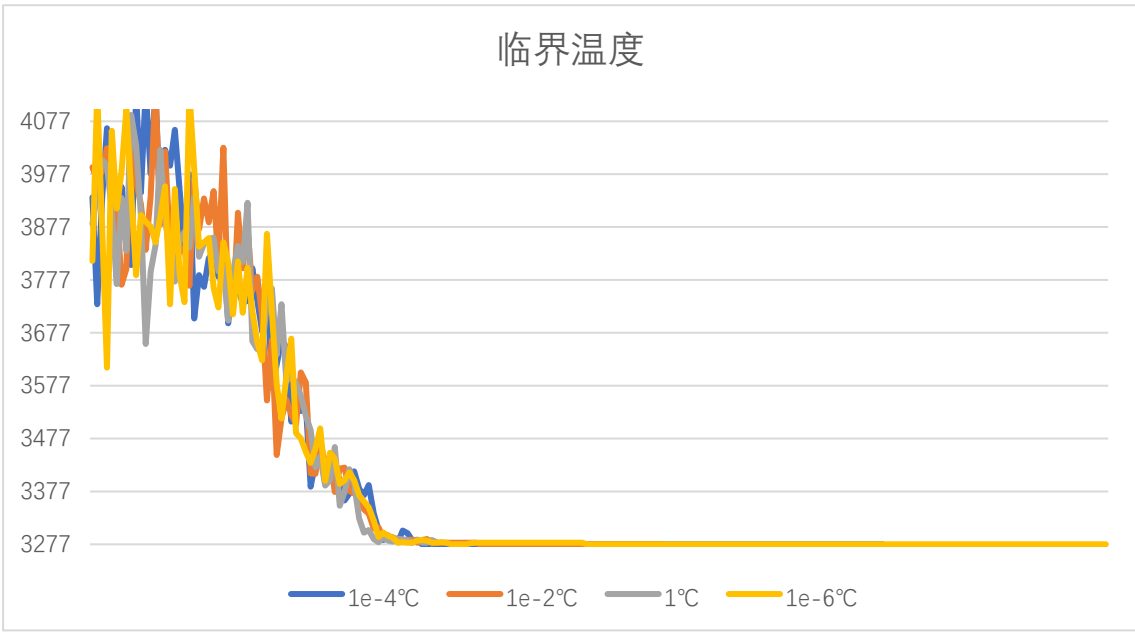
1500*0.99^1000=0.06,0.06*0.7 这个冗余度对计算路径太小了已经没什么意义。

结论：

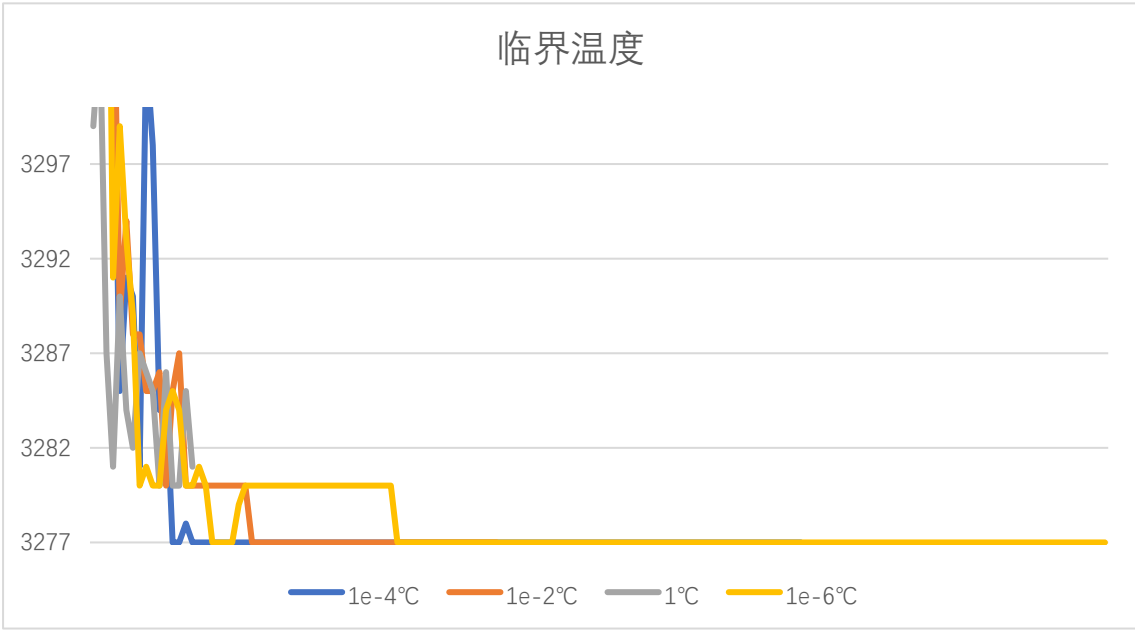
初始温度过低，退火过程短，很快就退化成登山算法，难以保证找到最优解；

初始温度过高，理论上初始温度越高，找到最优解的能力越强，但是退火过程太长，前期接受性太强，退化成随机交换，解的质量提升小，性能消耗大，计算时间长.

3.2.7 临界温度对模拟退火算法的影响



临界温度对模拟退火算法的影响



临界温度对模拟退火算法的影响（优化末期）

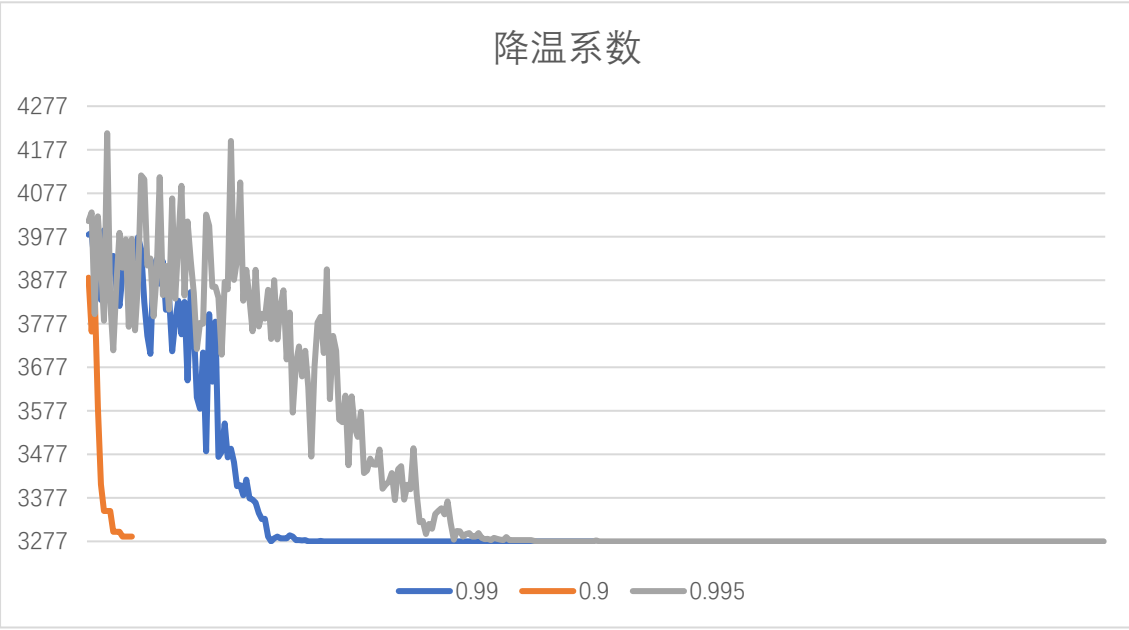
观察图表，在四组不同的临界温度参数设置下，除了 1℃组没有找到最优解，其余三组差不多同时找到最优解，而运行时长明显随着临界温度的降低而增大。1℃组最终给出的结果甚至不是运行中的最小值（最后接受了一个更差的结果）。其余三组根据其临界温度低的程度时长上有不同的延长。1e-6℃组的时长是 1e-2℃组的 3 倍。

结论：

临界温度过高，模拟退火算法过早结束，很难保证得到最优解，甚至无法确保得到的结果是计算出来的最好结果，因为温度高时比较可能会接受较差的结果。

临界温度过低，对较差结果的接受度低，退化成登山算法，由于登山算法的局限性，很难通过一次局部的变化优化结果，即便还没找到最优解也很难再在低温状态下通过登山算法找到最优解。应该通过调节其他参数而不是过渡地调低临界温度来提高准确率，因为对于很快找到最优解的用例来说，其在等待温度降低的循环中耗费大量不必要的时间和算力而不能改进任何结果。

3.2.8 降温系数对模拟退火算法的影响



降温系数对模拟退火算法的影响

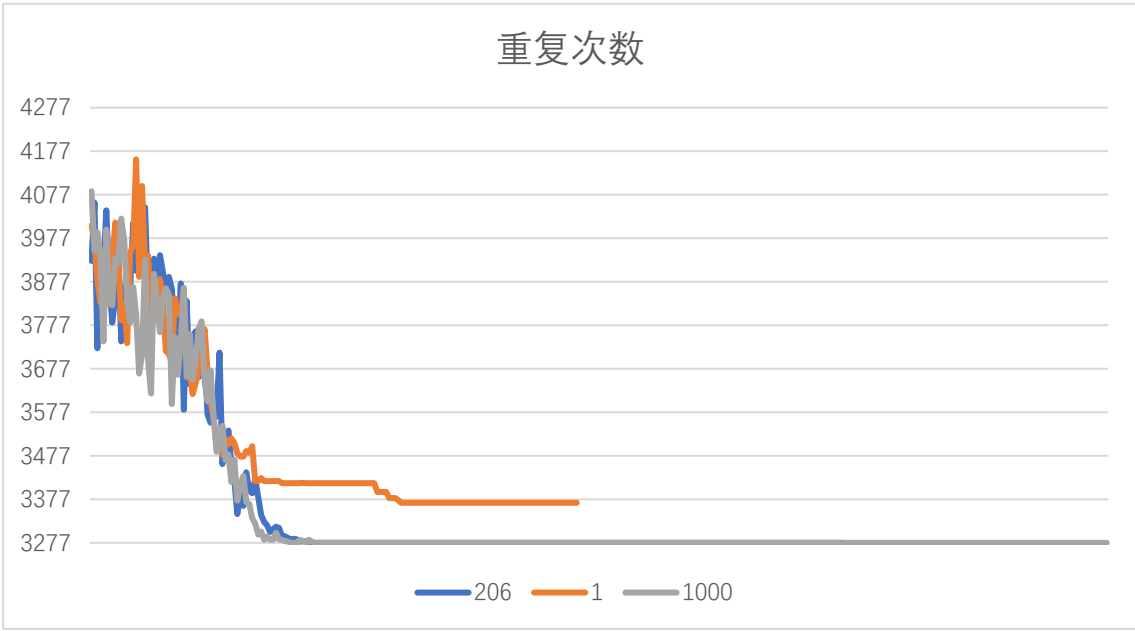
观察图表，降温系数直接影响模拟退火算法的进程，降温系数小时，温度下降剧烈，呈跳跃式指数下降，解的质量也快速提高，但很快就达到了临界温度。降温系数越接近于 1，温度下降越平滑，越趋近于连续下降，越趋近于物理上的准静态热力学过程。此外降温系数有效位数也直接影响了计算的运算量。过于趋近于 1 的降温系数，还要额外承担多出的有效位数的计算量。

结论：

降温系数过小，结果早熟，能很快向最优解收敛，但是还没找到最优解就停止了。

降温系数过大，由于指数的增长率，过大的降温系数使得模拟退火算法运行时间会以数量级的比例增长，理论上降温系数越接近于 1，越能模拟从初始温度以准静态过程缓慢得向外界散热降温至临界温度，结合模拟退火算法提出的背景，降温越平滑，结果质量越好，但是实际运用时要考虑运算量和计算时间的约束。

3.2.9　同一温度下重复次数对模拟退火算法的影响



重复次数对模拟退火算法的影响

由于模拟退火算法过程中温度的变化是乘以降温系数得到的，意味着温度以指数速率下降，为了避免温度下降过快导致程序早早结束，也为了避免通过牵强地修改初始温度，降温系数，临界温度造成模拟退火算法的头部和尾部出现不同程度的退化（如初期退化成随机交换，末期退化成登山算法），为了使以上参数处于一个数学上合理的范围，引入另一个参数——每个温度下重复操作的次数。它的值在物理上没有相对应的概念，其存在是为了弥补计算量的不足，其值的大小受初始温度，临界温度，降温系数的值的影响。其值用于将计算过程均匀地拉伸，可以给模拟退火算法的各个过程提供更多找到最优解的机会。

结论：

重复次数过小，程序过早结束，温度以指数速率下降，结果早熟，很难找到最优解。

重复次数过大，重复计算，尤其在运行末期，大量地进行退化后的登山算法，对解的质量提升甚微，但显著增加了算力消耗。

3.2.10　参数设置总结

综上，理论上初始温度越高，降温系数越趋近于 1，临界温度越低，越趋近于物理上的退火模型，解的质量越好，但是从实际上使用分析，过于极端的参数会使得模拟退火算法的初期和末期对解的贡献甚微，而过分地延长了计算时间，消耗了大量的算力。所以实际使用时要根据核心概率公式 exp(-delta / t)的数学含义，赋予参数合理的值，使得程序前期不会退化成随机交换，末期不会退化成登山算法，并在给定初始温度，降温系数，临界温度的条件下设置重复次数以弥补数学上导致的计算量不足的缺陷。在给定参数初始温度 T，降温系数 K，临界温度 C，可得降温次数 n 满足：

$$T * K^n = C;$$

引入重复次数 N 后，总计算次数 n 满足：

$$n = N * log_K \frac{C}{T}。$$

本文中标准参数设置为 T=1500℃，K=0.99，C=1e-4，N=206，计算可得总计算次数为 n=338,680 次，该参数条件下计算量小的同时保证了极高的准确率，能通过比样例输入规模大得多的随机数据。

3.2.11 与遗传算法在时间和空间上消耗的简要比较

通过与遗传算法比较发现，退火算法计算速率高，准确率高，空间占用率低，各方面性能指标都优于遗传算法，由于手写的遗传算法只实现了功能，还未进行调参，功能重构，并行化处理，运行时间和计算结果没办法横向比较。下图给出遗传算法在用例 0 上随机一次的优化过程，基线 7038 是退火算法给出的最优解，取值为每一代的染色体中适应值最高的个体的初始适应值。本报告不赘述遗传算法具体细节，具体细节见附带文档。遗传算法简要报告.pdf

由于本文写的遗传算法虽然功能完备但是没有调参，从图中可以看出还有较大进步空间，但是其长达数秒的计算性能表现显然是难以望模拟退火算法其项背，退火算法平均只需要 0.1s，而且遗传算法需要创建大量对象，也需要大量空间资源，这两点是其算法性质决定的，与参数关系不大，凭此两点可见模拟退火算法优异的表现。

4 总结

本文对模拟退火算法和登山算法在解决单机调度问题上的多个方面进行了详尽的比较。介绍了模拟退火算法和登山算法的核心思路，算法实现，介绍了解决单机调度问题的一种解法，并用动态规划的方法用算法实现。比较了两种启发式算法在单机调度问题上的优化效果，分析了两种算法的区别和原因。在不同的参数设置下进行了不同的对照试验，分别就初始温度，临界温度，降温系数，重复次数的参数设置对模拟退火算法的影响进行了重复独立实验。从运行时间，计算量和解的质量两方面对参数设置进行了评估，从模拟退火算法的物理意义，数学意义出发，从优化性能，准确率角度考虑，最终确定了对 10 组用例泛用性比较好的一组参数。在大量重复独立实验下，用数据展现了模拟退火算法在单机调度问题上稳定，优异的寻找最优解的能力。最后设计了另一种启发式算法遗传算法与模拟退火算法做比较，模拟退火算法首先在时间和空间占用上明显优于遗传算法，其次模拟退火算法以较简洁的方式取得了比遗传算法表现更好的成绩。

本文本来要写的不足之处比如读写文件，并行化处理，效率上的优化都在不断的改进中解决了，单次运行的时间由第一天的 90 秒降低到平均 0.1 秒。

仍存在的不足之处：

1.在设计一开始，没有从面向对象的思想出发，java 写得像 C 语言，没有组织好面向对象的架构，比如处理并行线程的类被定义为内部类，违反设计规范，但是和外部耦合性太强，后面不好再做修改，解决方法只能重新写过。（所以在设计用于作对比的遗传算法时，从一开始就按照面向对象的设计规范）。

2.没有实现多个用例一起处理，在跑实验用例时要应该用例跑完改一下路径名才能跑下一个用例，在实验过程中觉得这种操作很不得体，但是由于其与模拟退火算法核心功能关系不大，一直没有改进。

3.仍使用手动调参。参数互相制约，影响，不仅影响解的质量还影响程序的运行时间。目前参数的值是我认为在正确率和速度上达到了较好的平衡，虽然不能保证每次都能得到正确的结果，但其速度能保证相同的时间，它能运行几十遍，而其中有正确结果的概率大大增加，在用例这样的输入规模下得到最优解的概率趋近于 1.但是这样的一组参数值是手动无意中发现的，无法证明没有更好的一组参数值，而且针对不同的输入，不同参数的性能也不一样，无法确认一组泛用性好的参数，或者找到几组参数用于自适应不同的输入规模。我设想的一个解决方案是用遗传算法解决模拟退火算法的参数设置问题，模拟退火算法参数对应于染色体上的基因，这些参数的结构体构成染色体，可在遗传算法中去寻找针对不同输入规模表现优秀的那一组参数值，或是使用深度学习神经网络去训练遍历参数值集合，给出表现良好的参数。但最大的困难是，使用遗传算法去寻找模拟退火算法的参数，意味着遗传算法是描述模拟退火算法的“元语言”，得嵌套两者，考虑到遗传算法复杂的各种控制算子，我没有继续进行尝试。

参考文献:

[1] 越民义,韩继业.n 个零件在 m 台机床上的加工顺序问题(Ⅰ)[J].中国科学,1975(05):462-470.