旅游最短路径规划的数学模型

摘要

本文研究的是旅游路径规划问题，将路径规划问题使用图论抽象为数学模型，并计算不同条件下的最优路径。

针对问题一，本文采用了排列组合的方法，使用程序对所有可能路径进行穷举，最后得出最短路径为 1810 米。问题二是单人最优游览路径问题，在考虑步行时间、景点游览时间及开放时间约束的情况下，本文采用遗传算法求解游览顺序，以最大化游览时间，并引入锦标赛机制加快收敛速度，最后得到总游览时间为 270.745 分钟。问题三是在景点容量受限的情况下，规划三个旅游团的游览路径，针对该问题，改进了第二问的遗传算法进行求解，引入了惩罚机制，以快速求解不同旅行团的路径，最后得到每个旅行团的总游览时间分别为 255, 262, 266 分钟。问题四在第三问的基础上，推广至多个出发时间不同的旅行团，在该问题中通过调整惩罚系数，使算法遵守题目硬约束的情况下，尽可能能得出解。并且增加了算法的迭代次数，防止在复杂约束情况下得不到解，最后提出了对现有算法的改进方向与思路。

在本文最后，对模型进行了分析与评价。实验结果表明，经过 10000 次迭代优化，遗传算法可在短时间内求得近似最优解，且误差小于 0.0017%，验证了模型的高效性和准确性。进一步分析表明，引入软约束优化适应度函数，可有效提升遗传算法在多旅行商问题上的收敛效率与解的可行性。但在面对较多约束条件时，需大幅增加迭代次数，以避免无法求解或生成不符合约束的解，因此仍有优化空间。

关键字： 路径规划, 旅行商问题, 启发式算法, 遗传算法

一、 问题提出及问题分析

1.1 问题提出

随着东莞市旅游业的不断发展，科学合理的旅游路线能够提升游客的游览体验，并且对于优化景区资源配置也有重要意义。以隐贤山庄为例，园区内设置八个景点，并且每个景点均有不同的开放时间段和建议游览时间，园区内的步行速度均按 $2km/h$ 计，园区内各景点之间的距离与开放时间在附录A中给出。基于上述条件，我们提出了四个问题:

问题一：最短路径问题。从景点 1 出发，依次游览各个景点（景点内无需停留，仅计算行走时间），最终到达景点 8。建立数学模型，计算出一条行走距离最短的路线，并且列出所经过的景点。

问题二：单人最优游览路径问题。假设在 12:00 从景点 1 出发，要求在 17:00 前到达景点 8。在游览过程中，每个景点有各自规定的游览时间区间和开放时间。建立数学模型，设计一条可以走遍所有景点且在每个景点的游览时间之和尽可能长的路线，最后计算出在各个环节上的时间耗时。

问题三：多个旅游团协同游览问题。假设有 3 个旅游团同时在 12:00 从景点 1 出发，要求在 17:00 前到达景点 8。在游览过程中，除了景点 1 和景点 8 之外，其他景点只能在同一时间内容纳一个旅游团，如果当前时间段内已经有一个旅游团在某一景点，则后到的旅游团必须等待直至上一个旅游团完成游览后才能进入。建立数学模型，分别为 3 个旅游团设计一条可以遍历所有景点，并且可以使各个旅游团在各个景点游览时间之和尽可能长的路线，并且计算各个环节上所消耗的时间。

问题四：多个旅游团分时游览问题。在第三问基础上，假设若干个旅游团的出发时间，并且要求各个旅游团在 5 个小时内到达景点 8。在保持题目 3 的约束不变的情况下，建立数学模型，分别为各个旅游团设计一条可以遍历所有景点，并且可以使各个旅游团在各个景点游览时间之和尽可能长的路线，并且计算各个环节上所消耗的时间。

1.2 问题分析

隐贤山庄内有八个景点，并且每个景点的停留时间范围和开放时间不同。本文将通过建立数学模型，解决在不同条件下的游览路径规划问题。

针对问题一中的最短路径问题，在数据量较小的情况下，我们可以将其视为一个排列组合问题。通过对景点的全排列，使用穷举法计算出所有可能路径的总距离，然后选择最小距离作为解，最后可以得出最短路径。

针对问题二可以抽象为一个带时间窗约束的旅行商问题 [1]，假设在 12:00 于景点 1 出发，在 17:00 前到达景点 8，并且需要将景点游览时间区间与开放时长纳入考虑，在以上约束下使游览时间之和最大。

针对第三问可以看作是一个带有时间窗口和冲突约束的旅行商问题。在第二问基础上引入了多旅游团同时出发的限制，并且规定在同一时间内只能容纳一个旅游团。假如后续旅游团到达某景点时，前一个旅游团未完成游览，则该旅游团必须等待至前一个旅游团完成游览后才能进入。在本问中除了原本的步行时间和游览时间之外，等待时间也是一个影响整体行程规划的关键因素，需对各景点的游览时间和等待时间进行综合调度，以尽可能延长个旅游团的游览时间。

针对第四问可以看作对 n 个旅行商的调度问题，需在时间窗口和景点容量约束下，最大化各团的总游览时间。此时旅游团之间的排队和等待关系将更加复杂，需要考虑多个旅游团之间的排队和等待关系，以及景点容量的限制。并且对于各团游览时间进行优化，使所有旅游团的停留时间尽可能达到最长。

二、 模型假设

为了简化问题，并使模型更具普适性，我们作出以下假设:

1. 假设步行速度统一设定为 2千米/小时，不考虑游客个体差异。
2. 景点间步行距离遵循直线距离，不考虑特殊地形或绕行情况。
3. 不考虑其他游客的干扰，即不存在道路拥堵或通行限制。
4. 假设游客在每个景点的游览时间遵守题目给定的数据。
5. 假设所有游客都严格遵守景区的开放时间。
6. 游客可以准确遵循计划的行程安排，不会在非规划路径上停留或绕路。

三、 符号说明

符号	说明
$n = 8$	共有 8 个景点
m	旅游团的数量
d_{ij}	景点 i 到景点 j 的距离
T_i	游客到达景点 i 的时间
W_i	游客在景点 i 的游览时间
L_i	游客离开景点 i 的时间
S	任意景点子集（不包括起点和终点）
$O_{\text{open},i}$	景点 i 的开放时间
$O_{\text{close},i}$	景点 i 的关闭时间
$T_{\text{min},i}$	景点 i 的最小游览时间
$T_{\text{max},i}$	景点 i 的最大游览时间
v_{walk}	游客步行速度
$T_i^{(k)}$	团队 k 到达景点 i 的时间
$L_i^{(k)}$	团队 k 离开景点 i 的时间
$W_i^{(k)}$	团队 k 在景点 i 的游览时间
$x_{ij} = \begin{cases} 1, & \text{如果路径从景点} i \text{ 到} j \\ 0, & \text{否则} \end{cases}$	决策变量

四、 问题一的模型建立和求解

4.1 模型建立

对于问题一，可以抽象为一个带起点和终点约束的旅行商问题。我们将实际的地形视为一个无向图，景点为图的顶点，景点之间的距离 (详细距离见附录A) 为边的权重。由景点 1 出发，每个景点必须访问一次，最后到达景点 8。

目标函数:

$$\min \sum_{i=1}^{n-1} \sum_{j=2}^{n} d_{ij} \cdot x_{ij}$$

约束条件:

$$\sum_{j=2}^{n} x_{1j} = 1 \tag{1}$$

$$\sum_{i=1}^{n-1} x_{i8} = 1 \tag{2}$$

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1, \forall i \in \{2, 3, ..., 7\} \tag{3}$$

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1, \forall j \in \{2, 3, ..., 7\} \tag{4}$$

$$\sum_{i,j\in S} x_{ij} \leq |S| - 1, \forall S \subset \{1, 2, ..., n\}, 2 \leq |S| \leq n - 1 \tag{5}$$

目标函数表示最小化总路径长度，式(1)和式(2)确保路径从景点 1 开始到景点 8 结束，式(3)和式(4)确保每个中间景点只有一次进入和一次离开，式(5)消除子回路。

4.2 模型求解

根据以上表述以及问题需求，由于本问题中景点数量较少，我们可以采用穷举法来求解。由于起点和终点固定，我们需要对剩余的 6 个景点进行全排列，共有 $A_6^6 = 720$ 种可能的路径。我们可以计算每条路径的总距离，并选择其中最短的路径。

通过编程计算（代码见附录B），我们不难得出最终的最短路径为:

$$1 \to 4 \to 6 \to 2 \to 3 \to 5 \to 7 \to 8 \qquad \text{总距离为 1810 米}$$

五、 问题二的模型建立和求解

5.1 模型建立

目标函数:

$$\max \sum_{i=1}^{n-1} W_i$$

约束条件 (仅列出与上问不同处):

$$T_1 = 12 \times 60 = 720 \tag{6}$$

$$L_i = T_i + W_i, \forall i \in \{1, 2, ..., n\} \tag{7}$$

$$T_j = L_i + \frac{d_{ij}}{v_{walk}} \cdot x_{ij}, \forall i, j \in \{1, 2, ..., n\}, i \neq j \tag{8}$$

$$O_{open,i} \leq T_i \leq O_{close,i}, \forall i \in \{1, 2, ..., n\} \tag{9}$$

$$T_{min,i} \leq W_i \leq T_{max,i}, \forall i \in \{1, 2, ..., n\} \tag{10}$$

$$T_i + W_i \leq O_{close,i}, \forall i \in \{1, 2, ..., n\} \tag{11}$$

$$T_4 = 30 \times \left\lceil \frac{T_4}{30} \right\rceil \tag{12}$$

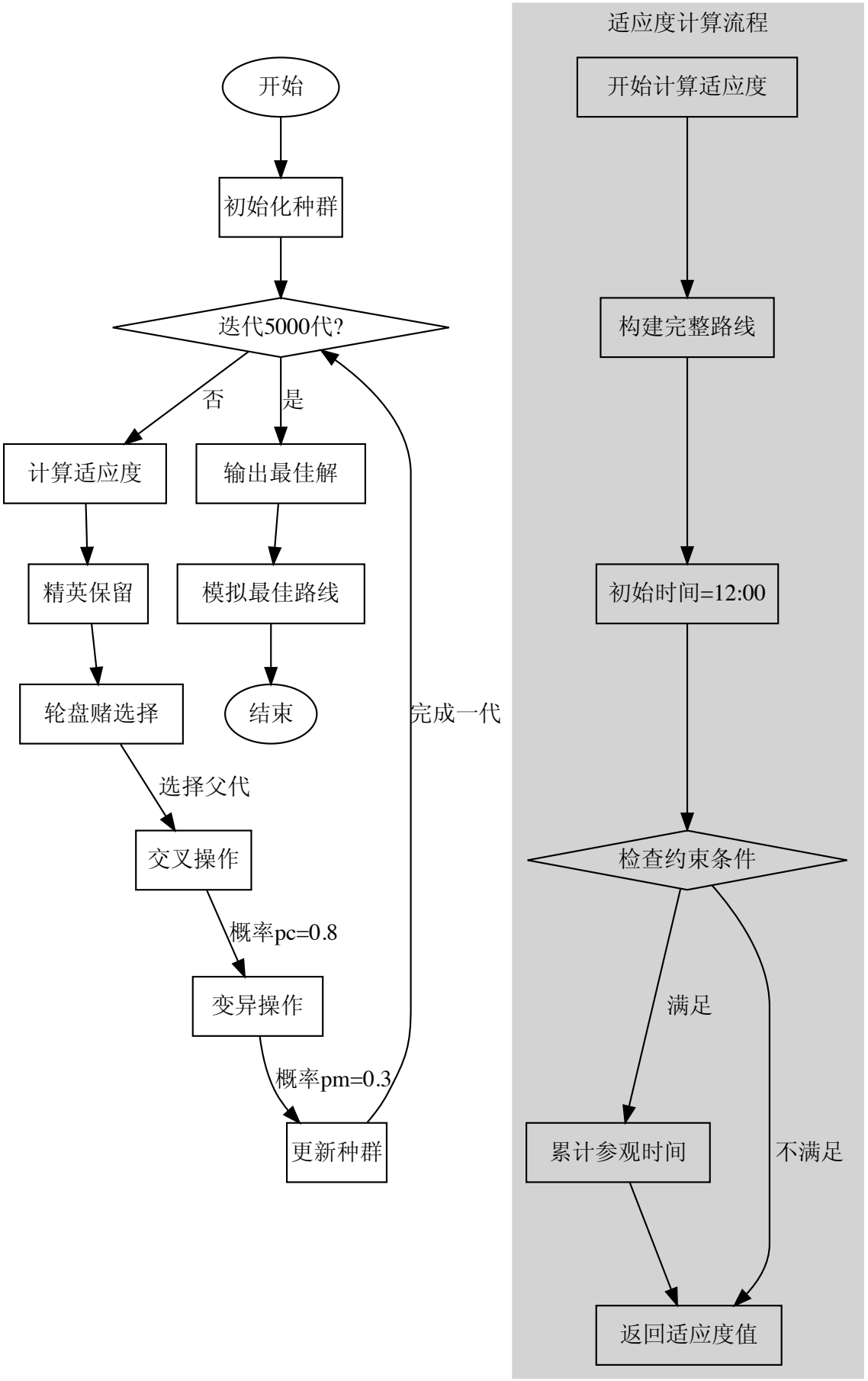
$$T_8 \leq 1020 \tag{13}$$

目标函数表示最大化总游览时间，式(6)定义起始时间为 12:00，式(7)和式(8)定义了离开时间和到达时间之间的关系，式(9)到式(11)确保在景点开放时间内到达和离开，式(12)针对景点 4 的整点半点入场做出约束，式(13)对到达景点 8 的时间做出约束。

5.2 模型求解

随着约束条件的增加以及数据规模的不断增长，传统精确解求解方法的计算复杂度呈现出指数级的增长趋势。因此，在面对大规模问题时，动态规划或混合整数规划等传统方法可能无法在合理的时间范围内提供精确解，所以遗传算法以及其他启发式算法在求解效率上会逐渐占据优势，从而为解决这类复杂问题提供了一个非常有前景的方法。

对于该问题采用了遗传算法 [2][3]，以应对复杂问题中穷举法的不适用性。遗传算法对于每个解 x_i，计算其适应度值 W_i。通过对当前解集进行操作，如交叉（将两个解 x_i 和 x_j 的部分路径重组）和变异（随机改变解 x_i 中的某一段路径），生成新的解集。根据适应度 W_i 的大小进行选择，保持总体规模不变，同时使得 W_i 值较高的解有更高概率被保留并参与下一轮迭代。经过多次迭代后，解集中的 W_i 整体提高，最终收敛到接近最优的解决方案。遗传算法的基本步骤概如下图所示。


开始
初始化种群
迭代5000代?
否
是
计算适应度
输出最佳解
精英保留
模拟最佳路线
轮盘赌选择
结束
选择父代
交叉操作
概率pc=0.8
变异操作
概率pm=0.3
更新种群
完成一代
适应度计算流程
开始计算适应度
构建完整路线
初始时间=12:00
检查约束条件
满足
不满足
累计参观时间
返回适应度值

遗传算法已经被广泛运用于寻找旅行商问题的近似最优解，旅行商问题在数据量大的情况下难以得到精确解，因此使用遗传算法求解旅行商问题是一个较优的选择。而在本文解决该问题时，使用的遗传算法经过改善，为了更快地收敛加入了锦标赛机制。该机制通过根据解的适应度对解进行排序，直接将优解放入子代解集中以避免解的重组和随机改变导致解集失去优解从而达到加速收敛的目的。

在题目给定数据下，设定每代解集包含 200 个解，交叉概率和变异概率分别设为 0.8 和 0.3，每代选取前五个优解直接放入下一个新解集，迭代 10000 次得出以下结果。

表 1　表 2 第 2 问的结果

序号	景点名称	达到时间点	游览时间 (分钟)	离开时间点
1	景点 1	12:00	8.09 min	12:08
2	景点 3	12:19	45.67 min	13:05
3	景点 5	13:11	42.59 min	13:54
4	景点 2	14:01	21.54 min	14:22
5	景点 4	14:30	30.00 min	15:00
6	景点 6	15:08	50.87 min	15:59
7	景点 7	16:12	41.98 min	16:54
8	景点 8	17:00	30.00 min	17:30
总的游览时间:270.745 分钟				
总的步行时间:59.255 分钟				

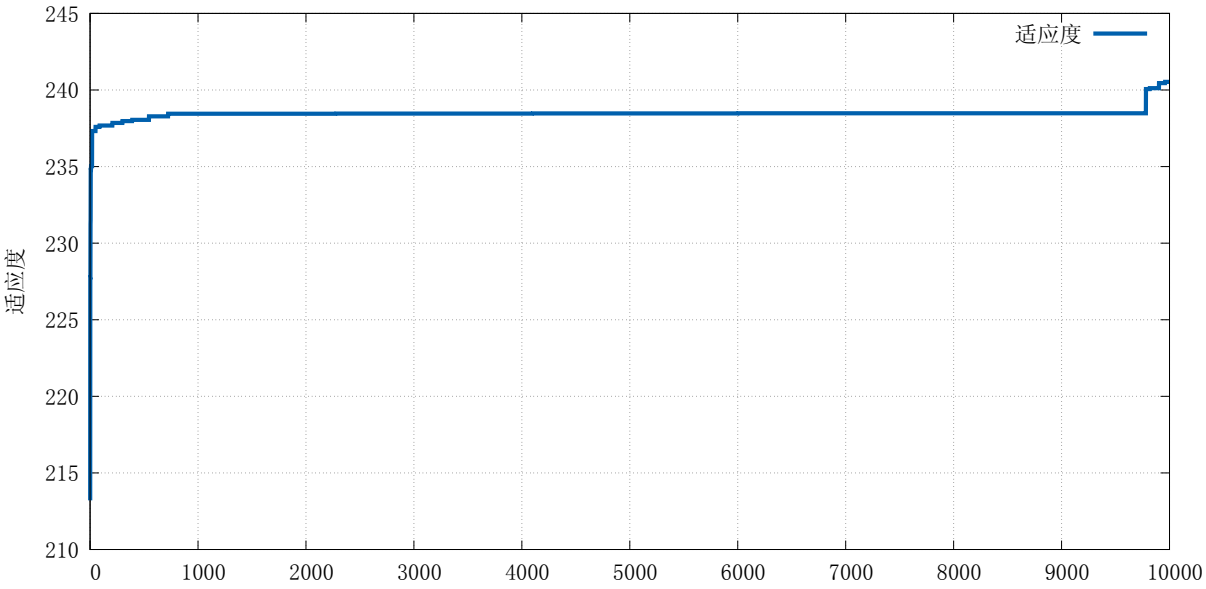


图 1　收敛曲线 (Gen=1w)

经过 10000 次的迭代，我们发现算法在较快时间内得到一个近似的最优解。这一结

果证明了经过优化的遗传算法在处理旅行商问题方面具有很高的实用性和可行性。

六、 问题三的模型建立和求解

6.1 模型建立

目标函数:

$$\max \sum_{k=1}^{3} \sum_{i=1}^{n-1} W_i^{(k)}$$

约束条件 (仅列出与上问不同处):

$$T_j^{(k)} \geq L_j^{(m)} - M(1 - z_j^{mk}) \quad \forall j \in \{2, \ldots, 7\}, \forall m, k \tag{14}$$

$$z_j^{mk} = \begin{cases} 1, \text{团队 } m \text{ 在团队 } k \text{ 之前访问景点 } j \\ 0, \text{否则} \end{cases} \tag{15}$$

式(14)表示的是等待规则。符号 M 表示一个足够大的正实数常数 M，它的作用是通过逻辑约束将离散条件（如二元变量 z_j^{mk} 的取值）与连续变量（如时间 $T_j^{(k)}$）联系起来，实现对复杂逻辑关系的数学建模。因为景点 2-7 只能容纳一个旅游团，若有多个旅游团，后到的必须等待。

因此在约束式 $T_j^{(k)} \geq L_j^{(m)} - M(1 - z_j^{mk})$ 中：当 $z_j^{mk} = 1$ 时（团队 m 在 k 之前访问景点 j）：

$$M(1 - z_j^{mk}) = 0 \implies T_j^{(k)} \geq L_j^{(m)}$$

此时约束生效，强制要求团队 k 必须等待团队 m 离开后才能进入景点 j。

当 $z_j^{mk} = 0$ 时（团队 m 不在 k 之前访问景点 j）：

$$M(1 - z_j^{mk}) = M \implies T_j^{(k)} \geq L_j^{(m)} - M$$

由于 M 极大，右侧 $L_j^{(m)} - M$ 会变为极小的负数，此时约束自然成立（因为时间 $T_j^{(k)} \geq 0$），相当于解除该约束。

6.2 模型求解

对比上一题，该问题对于题目约束更加严格，鉴于遗传算法的适应性和自学习性，在该题目依然使用遗传算法来解决。

虽然问题约束增加了，但是遗传算法大体的框架是不需要改变的。在此仅介绍该题目做出的改变。适应度函数是一个根据题目约束来编写的函数，因此题目约束改变了，计算函数应当跟随改变。在新的适应度函数中，适应度的计算分为两个部分组成:

- 总游览时间：所有旅游团在所有景点（包括出发点和终点）实际游览的总时间。
- 惩罚值：如果行程违反了一些规则（比如超时、容量冲突等），会根据违规的程度扣分。

并且在程序中引入了一个占用表来记录景点占用的时间段，通过这个值记录景点占用时段，并且增加惩罚值来主动引导算法避开，来防止旅行团在同一时段挤一个景点。在这里定义了四个惩罚系数：

1. 每分钟超时的惩罚：10 分。
2. 每分钟超过最后期限的惩罚：100 分。
3. 每分钟容量冲突的惩罚：20 分。
4. 每分钟入场时间偏差的惩罚：5 分。

对惩罚系数的定义做如下解释。超时惩罚的作用是约束算法避开部分景点超时的解决方案。而每分钟入场时间差的引入，则是为了约束模型使游客恰好在整点到达景点 4。因为这种等待是被允许的，所以他的惩罚度是相对较低的。超过最后期限的惩罚则是最严重的，因为超过了最后期限则代表该种群不能完成任务，所以要优先淘汰掉不能完成任务的，然后再根据具体表现再逐渐筛选出优秀个体。

针对本问题，延续使用上一问的参数配置下，将步行时间均向上取整，以保证解可行。算法经过 10000 代的迭代，最终获得了以下结果。

表 2　第 3 问结果

第 1 旅游团				第 2 旅游团				第 3 旅游团			
景点	到达	游览	离开	景点	到达	游览	离开	景点	到达	游览	离开
1	12:00	5	12:05	1	12:00	5	12:05	1	12:00	5	12:05
7	12:20	30	12:50	3	12:16	20	12:36	6	12:20	32	12:52
2	12:56	10	13:06	5	12:43	55	13:38	4	13:00	30	13:30
6	13:15	37	13:52	7	13:46	60	14:46	2	13:38	10	13:48
4	14:00	30	14:30	2	14:52	30	15:22	3	14:00	39	14:39
3	14:46	53	15:39	4	15:30	30	16:00	5	14:46	60	15:46
5	15:46	60	16:46	6	16:08	32	16:40	7	15:54	60	16:54
8	17:00	30	17:30	8	17:00	30	17:30	8	17:00	30	17:30
总步行时间: 75 分钟				总步行时间: 68 分钟				总步行时间: 64 分钟			
总游览时间: 255 分钟				总游览时间: 262 分钟				总游览时间: 266 分钟			
总等待时间: 0 分钟				总等待时间: 0 分钟				总等待时间: 0 分钟			

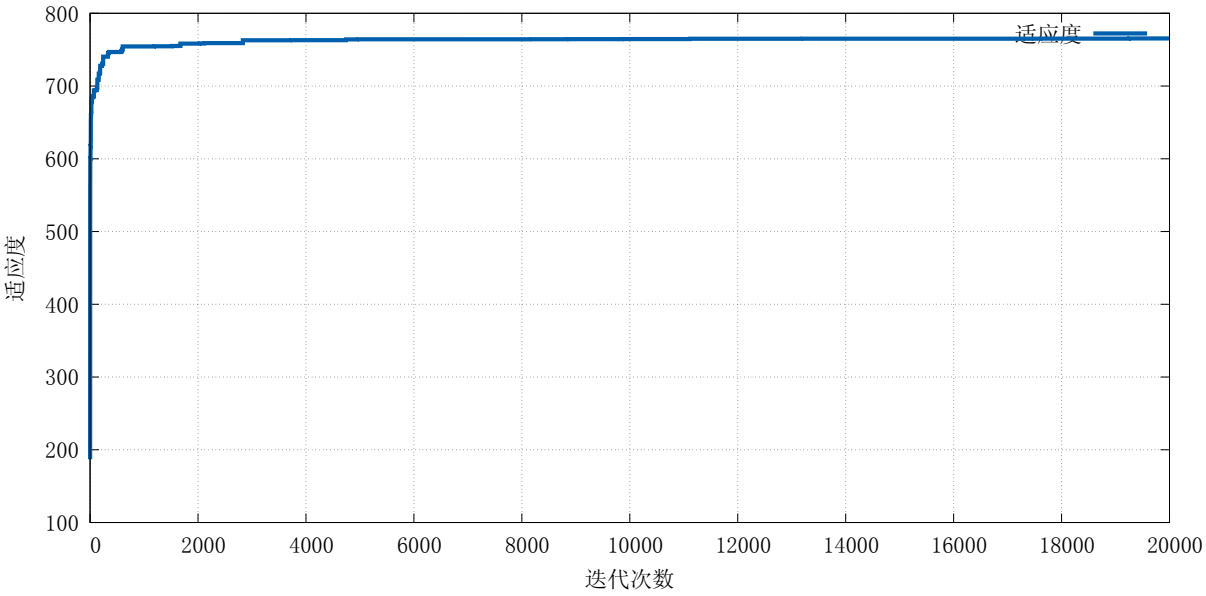


图 2　收敛曲线 (Gen=2w)

观察图(2)的收敛曲线可以发现，使用软约束的适应度函数后，可以更快地找到解开始迭代。因为遗传算法的特殊性，如果没有可行解将会一直随机生成直到出现一个可行解才开始迭代。而在这种约束比较严格的问题中，一直随机产生解集可能会出现永远无法得到第一个解的极端情况，因此使用软约束以允许在一开始出现的解不是最优解，来提高算法的全局搜索能力。后续可以通过增加迭代次数来贴合最优解。

七、 问题四的模型建立和求解

对比上一问，本问题仅改变了旅游团的出发时间和数量，因此约束不会有太大改变，较大改变部分是程序中对多旅游团不同时间出发做出适配。

目标函数:

$$\max \sum_{k=1}^{m} \sum_{i=1}^{n-1} W_i^{(k)}$$

在程序中不再通过硬编码定义旅游团的出发时间以及数量，并且降低了部分惩罚值来应对复杂约束的情况。通过增加迭代次数，依旧可以得到一个相对较好的结果。对于五个旅行团的情况，我们设置各旅行团出发时间分别为: 12:00、12:10、12:25、12:30、12:50，并且避免精度损失导致拟合变慢，将停留时间四舍五入与步行时间向上取整以确保解的可行性。经过 20w 次迭代，得到以下结果。

表 3　旅游团 1 行程表 (出发时间: 12:00)

景点	到达时间	离开时间	停留时间
1	12:00	12:05	5.0 min
2	12:15	12:25	10.0 min
7	12:31	13:26	55.0 min
5	13:34	14:07	33.0 min
3	14:14	15:14	60.0 min
4	15:30	16:00	30.0 min
6	16:08	16:38	30.0 min
8	17:00	17:30	30.0 min
路线: 1 → 2 → 7 → 5 → 3 → 4 → 6 → 8			
总游览时间: 253.0 min, 等待时间: 0.0 min, 步行时间: 75.0 min			
行程结束时间: 17:30			

表 4　旅游团 2 行程表 (出发时间: 12:10)

景点	到达时间	离开时间	停留时间
1	12:10	12:15	5.0 min
3	12:26	12:46	20.0 min
5	12:53	13:34	41.0 min
6	13:50	14:50	60.0 min
4	15:00	15:30	30.0 min
2	15:38	15:58	20.0 min
7	16:04	17:04	60.0 min
8	17:10	17:40	30.0 min
路线: 1 → 3 → 5 → 6 → 4 → 2 → 7 → 8			
总游览时间: 266.0 min, 等待时间: 0.0 min, 步行时间: 62.0 min			
行程结束时间: 17:40			

表 5　旅游团 3 行程表 (出发时间: 12:25)

景点	到达时间	离开时间	停留时间
1	12:25	12:35	10.0 min
6	12:50	13:50	60.0 min
4	14:00	14:30	30.0 min
2	14:38	14:48	10.0 min
7	14:54	15:24	30.0 min
5	15:32	16:02	30.0 min
3	16:09	17:02	53.0 min
8	17:25	17:55	30.0 min
路线: 1 → 6 → 4 → 2 → 7 → 5 → 3 → 8			
总游览时间: 253.0 min, 等待时间: 0.0 min, 步行时间: 75.0 min			
行程结束时间: 17:55			

表 6　旅游团 4 行程表 (出发时间: 12:30)

景点	到达时间	离开时间	停留时间
1	12:30	12:40	10.0 min
2	12:50	13:20	30.0 min
7	13:26	13:59	33.0 min
5	14:07	15:07	60.0 min
3	15:14	15:44	30.0 min
4	16:00	16:30	30.0 min
6	16:38	17:10	32.0 min
8	17:30	18:00	30.0 min
路线: $1 \to 2 \to 7 \to 5 \to 3 \to 4 \to 6 \to 8$			
总游览时间: 255.0 min, 等待时间: 0.0 min, 步行时间: 75.0 min			
行程结束时间: 18:00			

表 7　旅游团 5 行程表 (出发时间: 12:50)

景点	到达时间	离开时间	停留时间
1	12:50	12:55	5.0 min
3	13:06	13:40	34.0 min
2	13:52	14:22	30.0 min
4	14:30	15:00	30.0 min
6	15:08	15:46	38.0 min
5	16:02	16:56	54.0 min
7	17:04	17:44	40.0 min
8	17:50	18:20	30.0 min
路线: $1 \to 3 \to 2 \to 4 \to 6 \to 5 \to 7 \to 8$			
总游览时间: 261.0 min, 等待时间: 0.0 min, 步行时间: 69.0 min			
行程结束时间: 18:20			

观察上述表格可以发现，在当前约束的情况下，降低惩罚系数和提高迭代次数依旧可以得到结果，仅在少数旅游团上出现了步行时间偏大的情况。

因为能力与时间有限，未能改进出能在较小迭代次数内求解的算法，在此提供可能的改进思路。对于该问题，因为遗传算法的核心思路是不会有太大改变的，我们应该着重思考关于景点繁忙的情况。在此提出一个全局拥堵处理机制与动态冲突监测机制。

全局拥堵处理机制思路如下:

1. 动态优先级分配

   先到先得：先出发的团享有路径优先权，减少后续团的冲突概率。

   关键景点分流：对高频冲突景点，在初始化时分散各团的访问时间段。

2. 路径重规划触发条件

   拥堵阈值：当某景点的等待时间超过预设阈值，触发路径重规划:

(a) 替代路径生成：跳过当前景点，后续再掉头返回 (应考虑是否值得跳过，假若跳过后再回来的时间长过等待时间，则继续等待，避免出现得不偿失的问题)。

(b) 时间补偿：在后续景点中压缩游览时间。

动态冲突监测机制思路如下:

1. 时间轴模拟法

   全局时间轴：建立全局时间线，记录每个景点被占用的时间段。

   冲突检测：当某团计划到达景点时，检查该时间段是否已被其他团占用：

   - 无冲突：直接安排进入。
   - 存在冲突：计算等待时间至前团离开，更新该团的到达时间，并调整后续行程。

2. 等待时间优化策略

   被动等待：在冲突景点处等待，占用后续景点的空闲时间。

   主动绕行：动态调整路径，跳过当前繁忙景点，后续返回访问（需确保所有景点被遍历）。

另外对于现有算法提出优化：

1. 贪心算法初始化种群，保证一开始就存在解，方便后续迭代，并且可以增强局部搜索能力。
2. 就近访问策略，优先选择距离当前景点较近的下一个景点，来减少步行时间的消耗。
3. 变邻域搜索算法，对适应度高的个体进行单独优化，交换相邻景点顺序，检查时间是否可行，并且主动避开关键景点的访问时段，避开拥堵高峰期。
4. 分层遗传算法，优先优化各团的旅游路径，各团优化完之后再合并在全局层面上的优化。
5. 在适应度计算中，取消对容量约束与结束时间的惩罚，转为硬约束，避免算法产生不符合条件的解 (可能会导致收敛速度过慢问题，应结合上述优化机制搭配使用)。
6. 最后可以采用并行计算，最大化利用本地计算资源，加快求解过程。

八、 结果分析与模型评价

8.1 结果分析

1. 准确性分析

   由于问题二规模较小，可以编写代码使用整数线性规划求解器 (MILP) 进行求解，但该方法对于规模较大的问题三和问题四已不再适用，在此仅对问题二得到的结果进行两种方法的误差分析。经过计算，MILP 求解器得出的最优游览时间为 270.75

分钟，而使用遗传算法计算的近似最优解为 270.7455 分钟，两者之间的误差仅为：

$$\text{误差} = 270.75 - 270.7455 = 0.0045\text{分钟}$$

该误差值极小，在实际应用过程中可以忽略不计。由此可见，遗传算法在本问题上的求解性能是极为优秀的，其解的准确性几乎等同于精确算法的最优解。

2. 健壮性分析

评价一个算法时，健壮性是一个关键指标，它衡量算法在不同情况下是否能保持稳定的性能。为了验证在本问题中的健壮性，我们进行了多次模拟，并调整遗传次数，以测试其对最优游览时间的影响。

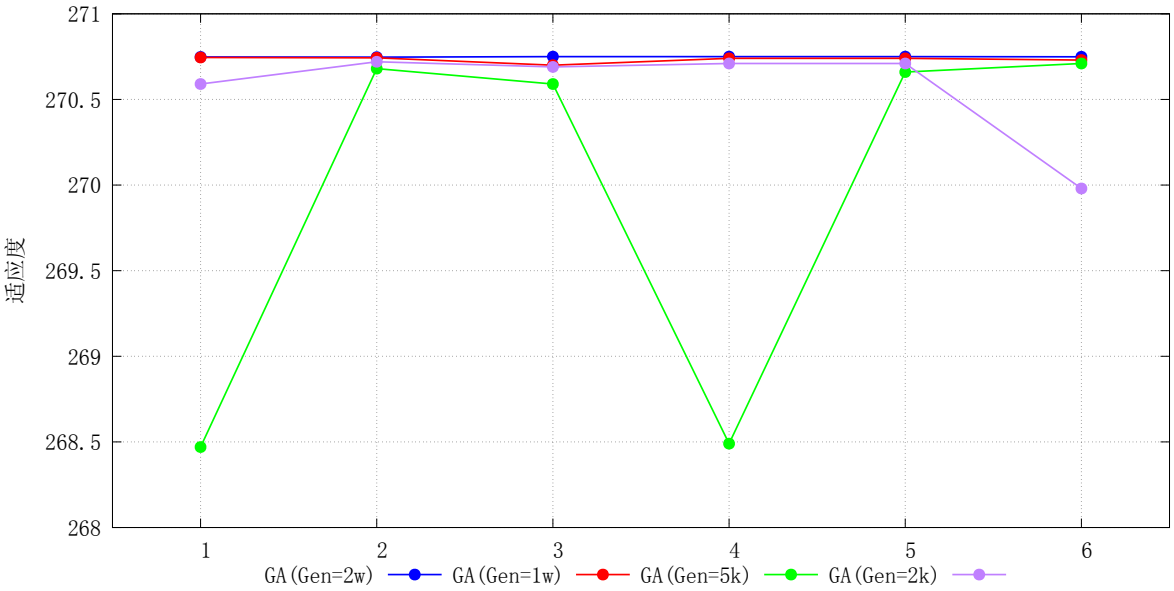


图 3　不同迭代次数下的表现

表 8　基因算法在不同代数下的性能比较

序号	GA(Gen=20000)	GA(Gen=10000)	GA(Gen=5000)	GA(Gen=2000)
1	270.748	270.745	268.47	270.59
2	270.747	270.743	270.68	270.72
3	270.75	270.7	270.59	270.69
4	270.75	270.74	268.49	270.71
5	270.75	270.74	270.66	270.71
6	270.749	270.73	270.71	269.98
均值	270.7490	270.7330	269.9333	270.5667
标准差	0.0013	0.0170	1.1265	0.2914
变异系数	0.0000	0.0001	0.0042	0.0011

根据以上两个图表可以看出在 20000 代下，算法的稳定性极高，误差极小（仅 0.001 分钟）。但是减少迭代次数会降低稳定性，但在 10000 代时仍保持良好效果。迭代 2000 代时，解的波动性增大，但是距离最优解仍然相差不到 0.3 分钟，但是求解时间仅是 20000 次的 $\frac{1}{10}$，因此该算法的健壮性是相当优秀的。

3. 灵敏性分析

在灵敏性分析方面，在适应度计算函数中使用的是刚性约束：

$$x_{ij} = \begin{cases} \sum \text{游览时间}, & \text{满足所有约束} \\ 0, & \text{不满足约束} \end{cases}$$

因此有违反题目约束的情况会直接判 0。让我们改变时间约束，稍微收紧和放松来观察算法表现，可以看到表(4)中在时间约束比较严格时，可能遗传算法不能立马找到可行解，需要迭代一段时间，存在改进空间。

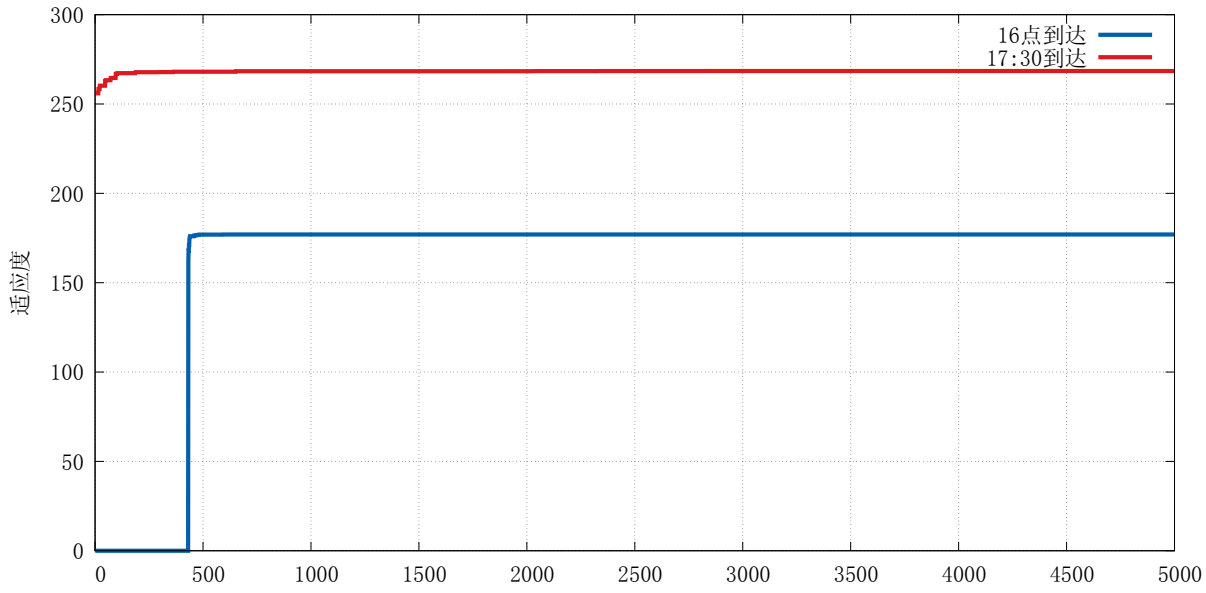


图 4　收敛曲线 (Gen=5k)

8.2 模型评价

在本次研究中，我们使用遗传算法进行旅游路径优化，目标是在有限时间内遍历所有景点，并最大化游览时间。为全面评估该模型的性能，我们从求解能力、计算效率两个方面对模型进行综合评价。

表 9　计算时间与误差分析

评价项目 / 求解方法	计算时间 (秒)	游览时间	绝对误差	相对误差
GA (GEN=2w)	3.5s	270.749	0.001	0.0004%
GA (GEN=1w)	1.8s	270.733	0.017	0.0063%
GA (GEN=5k)	1.2s	269.933	0.816	0.3016%
GA (GEN=2k)	0.4s	270.566	0.183	0.0677%
MILP	5.5s	270.75	0	0%

遗传算法在 2w 次的情况下已经比 MILP 快了将近 2s，而在更少的迭代次数时，其时间效率显著优于 MILP。随着题目难度提高，GA 适用于大规模问题的优势会逐渐显露，而 MILP 计算成本过高，不适用于实际应用。并且通过观察数据可以发现在 5k 迭代次数下的结果没有 2k 的好，这可能是因为在 5k 次迭代中随机生成的解比较差，导致迭代速度慢，而 2k 时迭代速度较快，导致 5k 次的结果甚至可能不如 2k 次。因此该模型存在改进空间以提高解的稳定性。

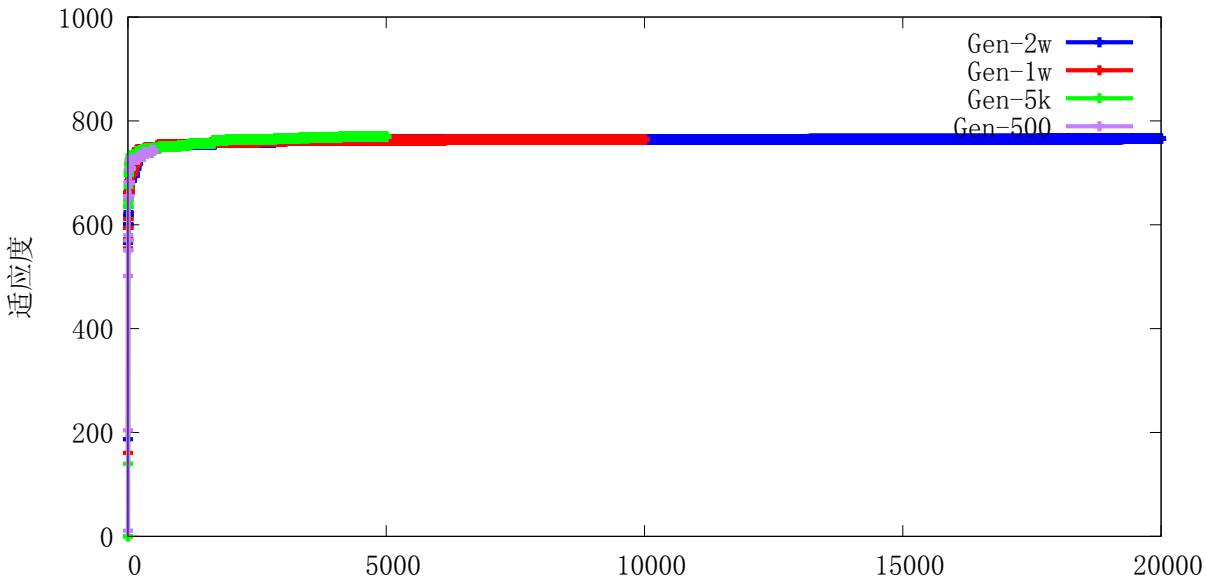


图 5　不同迭代次数下的表现

可以发现，在改为软约束的情况下，算法在迭代第 500 次时已经是相对较优地解了。因为在现实问题中，分钟级的精度已经是相当高的了，所以可以认为遗传算法在解决这种规模较大的问题的时候有着较大优势。

在求解问题四的过程中，发现随着约束条件的增加，找到初始解的难度提高，算法的迭代次数显著增加，并且迭代过程中有可能陷入局部最优解，因此需要更多的迭代次数才能找到全局最优解。在实际应用中，需要根据具体问题的规模和复杂度来选择合适

的算法和参数设置，并且引入优化机制来优化局部搜索能力，避免陷入局部最优解。由于遗传算法的随机性，可以看到图(6)中同一代码每次运行的结果都不一样，因此需要改进算法以提高稳定性与收敛速度。

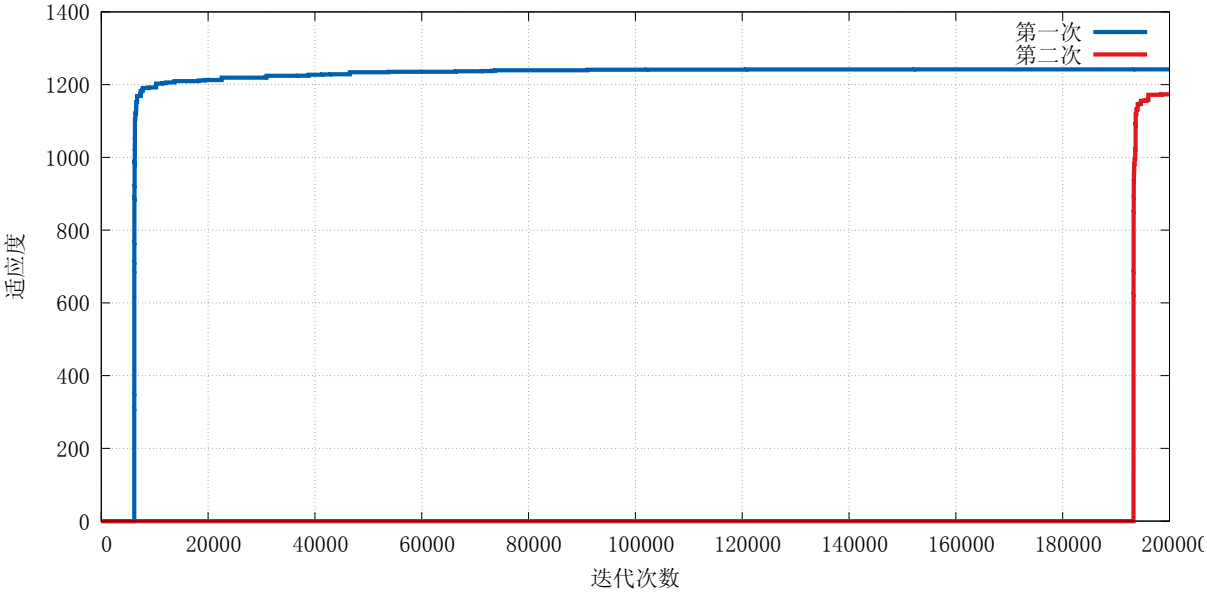


图 6　不同迭代次数下的表现

在实际应用中，需要根据具体问题的规模和复杂度来选择合适的算法和参数设置，并引入优化机制来优化局部搜索能力，避免陷入局部最优解。并且应当使用其他算法来辅助遗传算法，以解决随机搜索算法不稳定的问题，提高算法的全局搜索能力和稳定性。

参考文献

[1] 百度百科. 旅行商问题 [EB/OL]. [2025-03-01]. https://baike.baidu.com/item/%E6%97%85%E8%A1%8C%E5%95%86%E9%97%AE%E9%A2%98.

[2] 百度百科. 遗传算法 [EB/OL]. [2025-03-08]. https://baike.baidu.com/item/%E9%81%97%E4%BC%A0%E7%AE%97%E6%B3%95.

[3] 唐立新, 张煦, 李峰, 等. 旅行商问题 (TSP) 的改进遗传算法 [J]. 东北大学学报 (自然科学版), 1999, (1): 40–42.

[4] 姜群, 晏雨. 改进的遗传算法在 TSP 问题中的应用 [J]. 重庆理工大学学报 (自然科学), 2012, (9): 97–99.

[5] 高经维, 张煦, 李峰, 等. 求解 TSP 问题的遗传算法实现 [J]. 计算机时代, 2004, (2): 19–21.

[6] 郝艳伟, 李祖枢. 一种改进的遗传规划算法 [J]. 重庆理工大学学报 (自然科学版), 2011, (4): 74–78.

附录 A　题目附加信息

表 10　景点间距离矩阵

	景点 1	景点 2	景点 3	景点 4	景点 5	景点 6	景点 7	景点 8
景点 1	0	310	360	200	590	475	500	700
景点 2	310	0	380	250	230	285	200	390
景点 3	360	380	0	510	230	765	580	760
景点 4	200	250	510	0	480	265	450	640
景点 5	590	230	230	480	0	510	260	450
景点 6	475	285	765	265	510	0	450	640
景点 7	500	200	580	450	260	450	0	190
景点 8	700	390	760	640	450	640	190	0

表 11　景点游览时间与开放时间

景点	游览时间	开放时间
景点 1	5–10 分钟	9:00–16:00
景点 2	10–30 分钟	9:00–16:00
景点 3	20–60 分钟	9:00–18:00
景点 4	30 分钟	9:00, 9:30, 10:00, 10:30, 11:00, 11:30, 12:00, 12:30, 13:00, 13:30, 14:00, 14:30, 15:00, 15:30, 16:00, 16:30, 17:00, 17:30, 18:00（半点和整点游客才能进场）
景点 5	30–60 分钟	9:00–18:00
景点 6	20–60 分钟	9:00–18:00
景点 7	30–60 分钟	9:00–18:00
景点 8	30 分钟	9:00–21:30

附录 B　穷举法求解最短路径

```
def find_shortest(distances):
   n = len(distances)
   min = float("inf")
   shortest = None

   # 中间节点
   middle_nodes = list(range(1, n - 1))
   # 所有可能的路径
   all_possible_paths = itertools.permutations(middle_nodes)

   for path in all_possible_paths:
      full_path = [0] + list(path) + [n - 1]

      current_distance = 0
      for i in range(len(full_path) - 1):
         current_distance += distances[full_path[i]][full_path[i + 1]]

      if current_distance < min:
         min = current_distance
         shortest = [x + 1 for x in full_path]

   return min, shortest
```

附录 C　适应度计算

```cpp
double compute_fitness(const Cell &indiv) {
  vector<int> full_route = {1};
  full_route.insert(full_route.end(), indiv.route.begin(), indiv.route.end());
  full_route.push_back(8);
  double current_time = 12 * 60;
  double total_visit = 0.0;
  for (size_t i = 0; i < full_route.size() - 1; ++i) {
    int sp = full_route[i] - 1;
    if (current_time > CLOSING_TIME[sp]) {
      return 0.0;
    }
    if (sp + 1 == 4) {
      double m = fmod(current_time, 30);
      if (m != 0) {
        double wait_needed = 30 - m;
        current_time += wait_needed;
        if (current_time > CLOSING_TIME[sp]) {
          return 0.0;
        }
      }
    }
    double t_visit = clamp(indiv.times[sp], static_cast<double>(MIN_TIME[sp]),
                  static_cast<double>(MAX_TIME[sp]));
    if (current_time + t_visit > CLOSING_TIME[sp]) {
      return 0.0;
    }
    current_time += t_visit;
    total_visit += t_visit;
    int sp_next = full_route[i + 1] - 1;
    double d = DIST_MATRIX[sp][sp_next];
    double walk_min = d / SPEED_M_PER_MIN;
    if (sp_next + 1 == 8 && current_time + walk_min > DEADLINE) {
      return 0.0;
    }
    current_time += walk_min;
  }
  return total_visit;
}
```

附录 D　行程模拟

```
pair<vector<tuple<int, double, double>>, double>
simulate_route(const Cell &indiv) {
  vector<int> full_route = {1};
  full_route.insert(full_route.end(), indiv.route.begin(), indiv.route.end());
  full_route.push_back(8);
  double current_time = 12 * 60;
  vector<tuple<int, double, double>> schedule;

  for (size_t i = 0; i < full_route.size() - 1; ++i) {
  int sp = full_route[i] - 1;
  double arrival_time = current_time;

  if (sp + 1 == 4) {
    double m = fmod(current_time, 30);
    if (m != 0) {
    current_time += (30 - m);
    }
  }

  double t_visit = clamp(indiv.times[sp], static_cast<double>(MIN_TIME[sp]),
                static_cast<double>(MAX_TIME[sp]));
  double departure_time = current_time + t_visit;
  schedule.push_back({full_route[i], arrival_time, departure_time});
  current_time = departure_time;

  int sp_next = full_route[i + 1] - 1;
  double d = DIST_MATRIX[sp][sp_next];
  double travel_time = d / SPEED_M_PER_MIN;
  current_time += travel_time;
  }

  schedule.push_back({8, current_time, current_time + MIN_TIME[7]});
  return {schedule, current_time};
}
```

附录 E　PMX 交叉

```
pair<Cell, Cell> pmx_crossover(const Cell &p1, const Cell &p2) {
  int n = p1.route.size();
  uniform_int_distribution<> dis(0, n - 1);
  int c1 = dis(global_rng);
  int c2 = dis(global_rng);
  if (c1 > c2)
    swap(c1, c2);

  vector<int> child1_route(n, -1);
  vector<int> child2_route(n, -1);

  for (int i = c1; i <= c2; ++i) {
    child1_route[i] = p1.route[i];
    child2_route[i] = p2.route[i];
  }

  unordered_map<int, int> map1, map2;
  for (int i = c1; i <= c2; ++i) {
    map1[p1.route[i]] = p2.route[i];
    map2[p2.route[i]] = p1.route[i];
  }

  for (int i = 0; i < n; ++i) {
    if (i < c1 || i > c2) {
    int val = p2.route[i];
    while (find(child1_route.begin(), child1_route.end(), val) != child1_route.end()) {
      val = map1[val];
    }
    child1_route[i] = val;
    }
  }

  for (int i = 0; i < n; ++i) {
    if (i < c1 || i > c2) {
    int val = p1.route[i];
    while (find(child2_route.begin(), child2_route.end(), val) != child2_route.end()) {
      val = map2[val];
    }
    child2_route[i] = val;
    }
  }

  uniform_real_distribution<> dis2(0, 1);
  vector<double> child1_times(p1.times.size());
```

```
  vector<double> child2_times(p1.times.size());
  for (size_t i = 0; i < p1.times.size(); ++i) {
    if (dis2(global_rng) < 0.5) {
    child1_times[i] = p1.times[i];
    child2_times[i] = p2.times[i];
    } else {
    child1_times[i] = p2.times[i];
    child2_times[i] = p1.times[i];
    }
  }

  return {Cell{child1_route, child1_times}, Cell{child2_route, child2_times}};
}
```

附录 F　交叉与变异操作

```
pair<Cell, Cell> crossover(const Cell &parent1, const Cell &parent2) {
  auto [child1, child2] = pmx_crossover(parent1, parent2);
  uniform_real_distribution<> dis(0, 1);
  double alpha = dis(global_rng);
  for (size_t i = 0; i < child1.times.size(); ++i) {
  child1.times[i] = alpha * parent1.times[i] + (1 - alpha) * parent2.times[i];
  child2.times[i] = alpha * parent2.times[i] + (1 - alpha) * parent1.times[i];
  }
  return {child1, child2};
}
void mutation(Cell &indiv, double pm) {
  uniform_real_distribution<> dis(0, 1);
  int n = indiv.route.size();
  if (dis(global_rng) < pm) {
  uniform_int_distribution<> dis_idx(0, n - 1);
  int idx1 = dis_idx(global_rng);
  int idx2 = dis_idx(global_rng);
  swap(indiv.route[idx1], indiv.route[idx2]);
  }
  for (size_t i = 0; i < indiv.times.size(); ++i) {
  if (dis(global_rng) < pm) {
    uniform_real_distribution<> dis_time(MIN_TIME[i], MAX_TIME[i]);
    indiv.times[i] = dis_time(global_rng);
  }
  }
}
```