北京航空航天大学

MOBA 游戏位置适配机制的多约束分配问题

——随机优化技巧的应用

作者：19373260 张作舟 19373257 黄泽桓 19373180 郭瑞

（作者不分次序，贡献相同）

目录

摘要

本篇文章围绕着生活中的游戏玩家分配问题开展，由于系统中每位玩家的偏好及等候时长不同，为一局游戏分配这些玩家的位置和优先级也不尽相同。文章从构建数学模型的角度出发，通过引入偏好向量、可行域、分配集合表示等数学符号，构建损失函数等数学方法，将定性描述的实际问题转化为可以定量求解的数学问题。并重点围绕着爬山法、退火法、遗传法三种随机优化方法对多约束问题进行具体的分析和求解。最后，概括总结了此问题建模的步骤及方法，并对各个方法间的优劣性进行了分析比较。此外，还拓展总结了随机优化技巧在类似问题中的应用。

关键词：多约束分配 随机优化 爬山法 退火法 遗传算法 数学建模

一、问题描述

1.1 建模背景

本次建模问题的来源并非某次建模比赛、或是数学建模的某次习题，而是对生活中遇到的一种带约束分配问题产生了好奇。如今 MOBA（多人在线战术竞技游戏）的势头正热，我们常常会匹配网络上的玩家加入一场总共 10 人的游戏，而每个人又有着自己的位置偏好（游戏中每队常常有五种具有不同职能的位置，每名玩家可以在游戏开始前选择自己的第一偏好与第二偏好）；此外，每名玩家有着各自不同的已经等待时间。于是每一局游戏都会存在着带约束分配问题，如 1.2 所表述。

1.2 问题表述

现有一场总共 10 人的游戏对局，每队都有 5 个不同的位置。在当前时刻，系统中有 20 位等待了不同时长的玩家想要加入这场游戏，每名玩家有自己理想的第一位置和第二位置。部分玩家对第二位置没有严格要求。问如何挑选玩家并为他们安排合适的位置，才能使总体满意度达到最大。

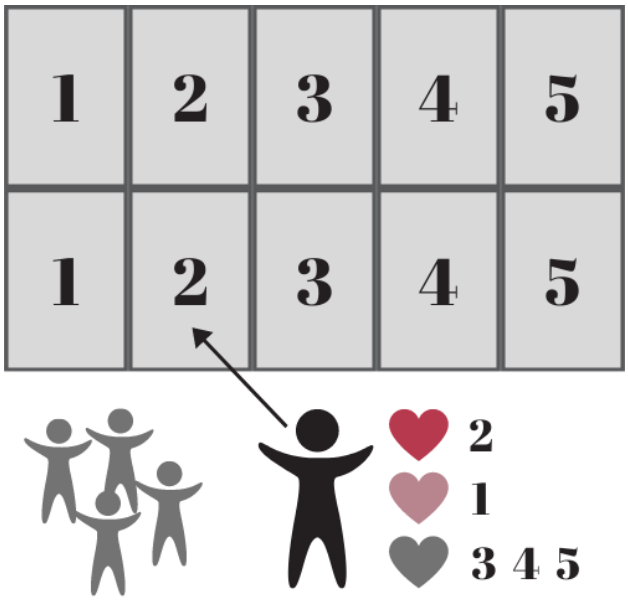

图 1 问题示意图

二、模型假设

1. 这 20 名玩家的实力相当，即分配时不考虑两队的能力水平不一致问题；
2. 随着等待时间的增长，玩家的耐心会越来越差；

3. 系统可以获取到玩家已等待的时长；

4. 每名玩家首选可以是五个位置中的一个；次选可以是除首选外的四个位置加上“补位”（玩家的第二偏好不强烈）共五个选择中的一个。

三、模型建立

3.1 问题抽象

表 1 符号定义表

符号	含义
i	唯一标识玩家的编号，范围为 1~20
t_i	玩家的等待时长（单位：分钟）,范围在 0~3min 之间
A_b~E_b　F	标识玩家偏好，A_b~E_b家偏标识五个位置，F为补位 $b=0/1$用以标识此位置属于哪一队
$pref_i=(X,Y)$	玩家的偏好向量，X为首选，Y为次选 $X\in[A_b,E_b]$，$Y\in[A_b,E_b]\cup\{F\}$，$X\neq Y$
$distri=(X_b,x_i)$	表征游戏内一个位置的分配，指为位置X_b分配玩家i
$domain$	所有可行分配方式组成的解空间
C	一种分配方式的总成本
C_1	一种分配方式的偏好成本
C_2	一种分配方式的等候时间成本

为将问题规范化、抽象化，我们整理出本文所使用的符号与意义如表 1，并对问题做出如下数字化的定义。

用i来为玩家的编号，取值范围应为 1~20。对于模型假设中的位置选定，我们规定A_b~E_b分别代表首选可以选择的五个位置，A_b~E_b、F为次选可以选择的六个位置。其中F代表补位，即玩家第二预期位置无指定；二进制角标b标识了此位置属于哪一队。此外，还设定一个变量t_i表示该玩家该时刻的等待时长。

依据模型假设和题设，重点分析玩家偏好的表示及取值情况，引入二元向量$pref_i=(X,Y)$，第一维表示首选位置，第二维表示次选位置，位置不能重复，符号化表示如下：

$$pref_i \in \{(X,Y)|X \in [A_b, E_b], Y \in [A_b, E_b] \cup \{F\}, X \neq Y\}$$

抽象化分配过程。定义元组(X_b, x_i)为一种分配，表示为位置X_b分配玩家i。可将队伍双方共 10 个待分配空缺位置抽象成大小为 10*1 的数组，数组元素为分配元组，初始状态为：

$$[(A_0, x), (A_1, x), (B_0, x), \dots, (E_0, x), (E_1, x)]$$

分配过程即依次为一个数组元组分配一名未分配玩家i，并将此元组从待分配的数组中移除至已分配的数组，重复此过程直至待分配数组的大小为空。例如此数组表征一种分配：

$$[(A_0, x_{12}), (A_1, x_2), (B_0, x_{20}), (B_1, x_6), (C_0, x_3), (C_1, x_{16}), (D_0, x_9), (D_1, x_4), (E_0, x_{17}), (E_1, x_1)]$$

显然这是一种有效的分配，但并不一定是我们所需要的最优化分配。为衡量一种分配的好坏，我们需要一种评判的尺度，即 3.2 抽象出的损失函数。

另外，由于在分配过程中，我们不必规定两边队伍的差异性，因此不妨设两支队伍的同个位置相同；且在将位置分配给玩家的过程中，每分配给一个玩家，后续将不可重复分给同一玩家，因此可设定在$x_1 \sim x_{20}$中将位置分配给x_1后，将x_1从可分配玩家中剔除，剩下玩家自动排序为$1 \sim 19$。故问题解的可行域可抽象为：

$$[\{A, (1,20)\}, \{A, (1,19)\}, \{B, (1,18)\}, \{B, (1,17)\}, \dots, \{E, (1,2)\}, \{E, (1,1)\}]$$

3.2 损失函数

为了衡量不同种分配方式的优劣，首先可以定义一个成本函数C，也即损失函数，一种合理的分配意为其有更低的函数值。分析题目以及模型假设的需求，影响C的变量分两个维度：玩家的位置偏好及等候时间。

位置偏好成本

显然为一名玩家分配了其理想位置不会导致总成本的增加。然而一局游戏中同一位置只有两名玩家的空缺，不可能满足所有人的首选是相当普遍的情况，甚至部分情况次选也无法满足，这些情况都会导致从成本的增加。据此我们人为假定一些参数，定义位置偏好成本函数C_1如下：

$$C_1 = \sum_{j=1}^{10} C_{1_j} \quad (1)$$

其中C_{1_j}满足（其中i为该分配位置上的玩家编号，$[n]$表示取某一向量的第分量）：

$$C_{1_j} = \begin{cases} 0, & destri_j[1] = pref_i[1] \\ 1, & destri_j[1] = pref_i[2] \\ 3, & destri_j[1] \notin pref_i \\ 2.5, & pref_i[2] = F \end{cases} \quad (2)$$

自然语言表述即：该位置是玩家的首选位置则不增加成本，次选位置成本加 1，其他位置则成本加 3，玩家次选为补位则成本加 2.5。

等候时间成本

为位置分配玩家时还需要考虑当前玩家所等候的时间。考虑到玩家的耐心随着时间增加损耗的速度也会增加，所以我们初步估计时间成本函数应该是非线性的。此外需要特别注意的是，为一名等候时间相对较短的玩家分配时会使总成本大量增加，而等候时间相对较长的玩家相反。这是游戏公平性的体现，系统会优先选择等待时间较长的玩家参与分配，因此其等候时间成本相对较小。我们初步刻画等候时间成本函数如下：

$$C_2 = \sum_{j=1}^{10} C_{2_j} \quad (3)$$

其中C_{2_j}满足（其中i为该分配位置上的玩家编号）：

$$C_{2_j} = 5e^{-t_i} \quad (4)$$

函数图像大致如下：

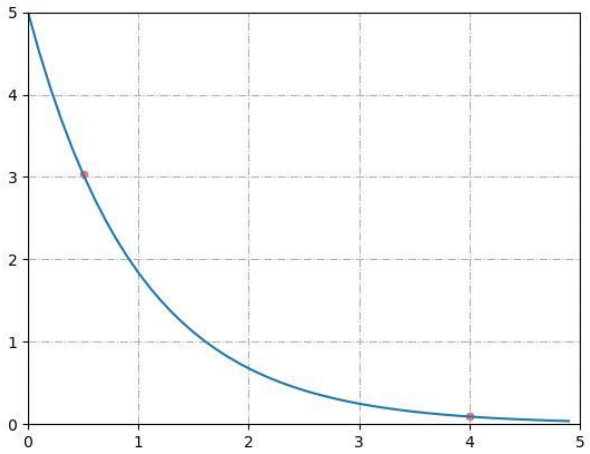

图 2 等候时间成本图

例如由图 2 可看出，玩家在等候了 30s 时，为空缺位置分配此玩家的等候时间成本为 3，而如果玩家等候时间超过了 4 分钟，则为其分配几乎不产生等候时间成本。

总成本——损失函数

总的成本函数，也即损失函数C是受C_1、C_2共同影响的，根据(1)(3)的结果，初步可令总损失函数为：

$$C = \lambda C_1 + \mu C_2 \quad (5)$$

对于每一种分配情况（可行解），代入损失函数(5)或(6)，即可量化其分配的优劣。损失函数伪代码如图 3 所示，具体实现代码见附录 1。至此建模工作完成。下面利用数学模型和损失函数，对此问题采取不同的随机算法进行求解。

算法 1 损失函数算法

输入: $Solution$ 当前解，$Prefs$ 玩家偏好，$Times$ 已等待时间

输出: $Cost$ 总成本

```
function Loss(Solution, Prefs, Times)
    Cost ← 0
    for each i ∈ Solution do
        cost0 ← cost0+ PrefCost(Prefs[i], Solution)
        cost1 ← cost1+ TimeCost(Times[i], Solution)
        cost ← λcost0 + μcost1
    end forreturn Cost
end function
function PrefCost(Pref, Solution)
    Cost ← 0
    if Solution[i] ∈ AbleAll then
        if Pref[0] then
            Cost ← Cost + 0
        else
            Cost ← Cost + 2.5
        end if
    else
        if pref[0] then
            Cost ← Cost + 0
        else if pref[1] then
            Cost ← Cost + 1
        else
            Cost ← Cost + 3
        end if
    end ifreturn Cost
end function
function TimeCost(Time, Solution)
    Cost ← λ/math.exp(Time) return Cost
end function
```

图 3 损失函数伪代码

四、模型求解

4.1 数据约束

为使题目问题具体化，我们参考了著名 MOBA 游戏 League of Legends 排位模式的玩

家位置偏好和等候时长信息，估计一般情况下的数据范围及频率，加权随机生成玩家的位置偏好及等候时间表如表 2 所示。之后给出的不同求解算法均针对表 2 数据进行运算。

表 2 玩家数据表

玩家 ID	第一偏好	第二偏好	已等候时间(s)
1	上路	中路	200
2	中路	上路	20
3	下路	补位	203
4	上路	中路	30
5	中路	辅助	50
6	辅助	中路	110
7	中路	下路	70
8	中路	下路	140
9	中路	辅助	50
10	上路	打野	240
11	上路	补位	120
12	中路	打野	200
13	上路	中路	2
14	辅助	补位	10
15	打野	中路	120
16	中路	打野	30
17	下路	中路	48
18	打野	补位	46
19	上单	中路	58
20	中路	上路	240

4.2 求解说明

求解此类多约束问题的可行域往往非常庞大，单是本题中可行域结果就有A_{20}^{10}种，简单的枚举算法在解决此问题时是不可能的。在处理复杂约束、多元变量的最优化问题时，我们通常可以采取随机优化算法，不强制要求找到全局最优解，而是不断优化有效解，尽可能高效地逼近全局最优解。

为高效获得本问题的一个极优解，我们主要采用了性能依次上升的爬山法、退火法和遗传算法，分别对实际问题依据已建立的数学模型进行分析求解，并探讨不同算法的性能和可靠性。

4.3 随机优化求解

4.3.1 爬山法

爬山法以一个随机解开始，在其附近的解集中寻找更好的题解。其算法流程大致为：获取一个随机解，随机选择当前解 10 个维度中的某个维度变量，判断该变量+1 或-1 后新解的损失函数值。若新值小于旧值，则用新解替换旧解再次迭代判断；若两个方向的新值都大于旧值，则迭代结束，当前极小值为最优解。

爬山法的合理性在于，充分利用了损失函数本身的连续可微性，每次尝试都建立在充分利用历史尝试的信息之上，逐渐找到一个局部最优解。

然而爬山法难以产生全局最优解。爬山法常常会限于局部范围内的最小值，它比临近解表现好，但却不是全局最优的。就本题而言，经过实践，爬山法常常在进行三四次迭代之后找到局部优解，即结束优化过程。最终得到的分配方案总成本较高，难以得到表现较优的解。

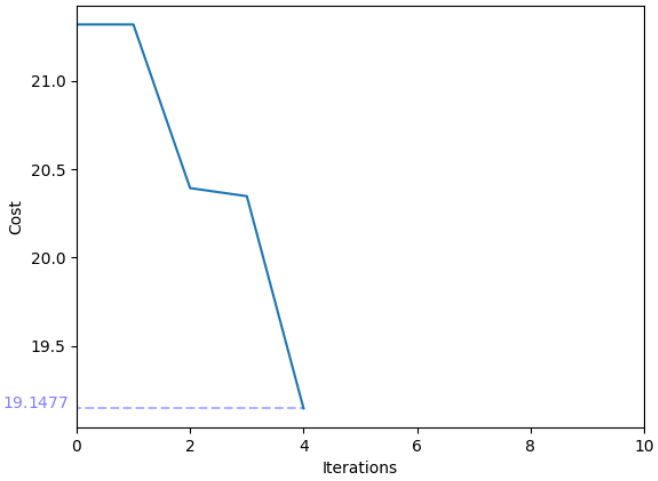

图 4 爬山过程中损失函数值变化图

4.3.2 模拟退火算法

模拟退火算法是受物理学的退火启发而来的优化算法。退火是将合金加热后慢慢冷却的过程，加热会使能量变大，大量的原子受到激发而向周围随机移动，冷却时速度较慢，使得原子有更多的机会找到一个比原先内能更低的位置[2]。模拟退火正是从一个随机的初始解开始不断的迭代，初始温度非常高，在迭代中不断降低。模拟退火算法流程如下：

1. 从可行域中随机获取初始解，进入迭代；
2. 在每次迭代中，算法使用随机性策略对当前解进行扰动。比较新解与旧解的损失：
   1) 若新解的成本值更低，则新解成为当前解
   2) 若新解的成本高，则按照下列退火跃迁公式计算概率，根据概率结果决定是否采取该解为当前解。该策略使题解有概率跳出局部最优解。可以看到初始时温度(T)非常高，概率几乎为 1，意味着迭代开始阶段大概率可采用较差解作为新

解。

$$p = e^{-(highcost-lowcost)/temperature}$$

3. 每次迭代，将以一定的冷却速率(cool)降低温度，直至温度接近 0 时，退火完毕，进入最后的局部渐进收敛过程。

本问题采取的扰动策略为，随机选择当前解 10 个维度中的某个维度变量，随机选择-1 或 1 与该变量相加。若得到的新解属于可行域，则采取其为新解；若不可行则取可行域的临界作为新解。解题算法详见图 5 伪代码，具体代码见附录 2。

算法 2 模拟退火算法

输入: $Domain$ 可行域，$costf$ 成本函数，$T0$ 起始温度，$T1$ 终止温度，$cool$ 冷却速率，$step$ 扰动幅度

输出: $Solution$ 最优解

```
function AnnealingOptimize(Domain, costf, T, cool, step)
    Solution ← random(Domain)
    while T0 > T1 do
        i ← random(Domain.size)
        j ← random(−step, step)
        Next ← Solution[i] + j
        p ← exp((−costf(Solution) − costf(Next))/T)
        if costf(Solution) > costf(Next) or p > random then
            Solution ← Next
        end if
        T ← T ∗ cool
    end while
    return Solution
end function
```

图 5 模拟退火算法伪代码

如图 6 优化过程展示，在退火过程的开始阶段，展示出损失函数值较高的解，而随着优化过程的不断进行，算法接受较差解的可能性大大降低，收敛逐渐稳定，最终得到总成本低、表现好的最优解（见表 3）。

表 3 模拟退火算法所得解

游戏位置	上路	中路	下路	打野	辅助
分配玩家	Player01	Player15	Player03	Player10	Player05
玩家偏好	上路，中路	打野，中路	下路，补位	上路，打野	中路，辅助
等待时间(s)	200	120	203	240	50
分配玩家	Player11	Player20	Player08	Player12	Player06
玩家偏好	上路，补位	中路，上路	中路，下路	中路，打野	辅助，中路
等待时间(s)	120	95	140	200	110

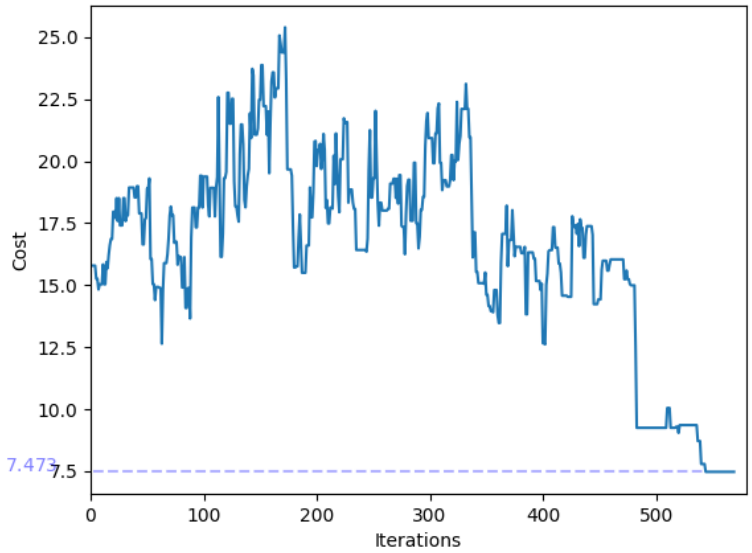

图 6 模拟退火过程中损失函数值变化图(T=10000, cool=0.98, step=1)

大多情况下，在得到一个更优的解之前，转向一个更差的解这一前序步骤是有必要的。若优化过程中获得一个优解后立即结束迭代，则很大可能会陷入局部最优解，而无法获得全局最优解。而模拟退火算法采用退火跃迁公式进行决策，在一定几率下允许较差解成为当前解，这是为了避免局限于局部最优的一种概率性尝试。

4.3.3 遗传算法

遗传算法也是受自然科学的启发，生物进化观点中有着自然选择、基因重组、基因突变这样的现象。同样的，随机优化算法也可以模拟自然选择过程，有效解之间相互借鉴优点（基因重组），并在结集中加以扰动（基因突变）来达到随机优化的目的。这类算法的运行过程具体如下：

1. 先在可行域内随机生成一组有效解，并利用损失函数计算每组解的成本；
2. 依据有效解的成本进行排序，并选出位于前列一定百分比的优胜解（自然选择）；
3. 对一定比例的优胜解的某一分量进行微小的、简单的、随机的改变（基因突变）；
4. 随机组合优胜解的分量并产生新的一组有效解（基因重组）；
5. 重复以上步骤直至达到迭代次数或有效解的成本值收敛于定值。
6. 取最后一代的最优解作为随机优化的最终解。

在本题中我们先随机选取 100 种可行的分配，利用每种分配的玩家索引构造字符串基因序列，如图 7 给出了分配和字符串基因序列的映射关系：

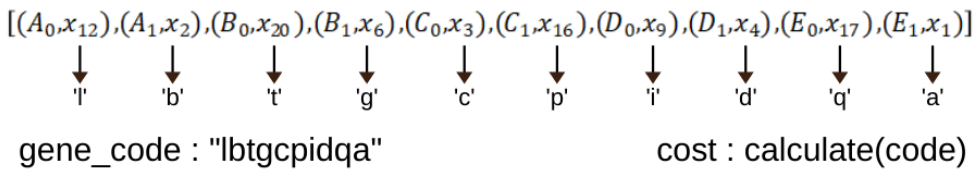

图 7 遗传算法中字符串基因序列的构建

至此，我们将每种分配在代码层次进一步抽象为字符串模型，可以根据不同编码的 *cost* 值排序并择优进入下一轮；设定合适的变异率和重组率，利用字符串的替换及拼接方法完成基因突变和基因重组的过程，不断产生 *cost* 更小的子代基因序列，重复以上操作。

迭代 50 次，取每一代的最优解计算损失函数值，并绘制迭代折线图如图 8 所示。

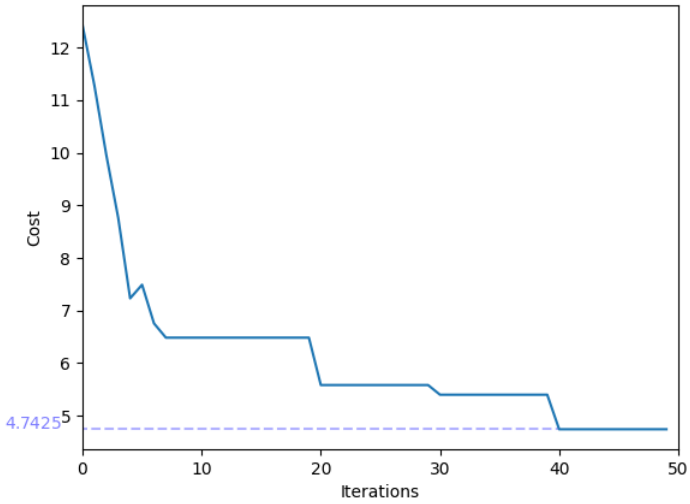

图 8 遗传算法过程中损失函数值变化图

由上图可见遗传算法的优化效果很好，已经将损失函数值降至 4.7425。最终解果的字符串基因序列为"jkcolthgaf"，即代表分配：

$$[(A_0,x_{10}),(A_1,x_{11}),(B_0,x_3),(B_1,x_{15})(C_0,x_{12})(C_1,x_{20})(D_0,x_8),(D_1,x_7),(E_0,x_1),(E_1,x_6)]$$

表 4 遗传算法所得解

游戏位置	上路	中路	下路	打野	辅助
分配玩家	Player10	Player12	Player08	Player03	Player01
玩家偏好	上路，打野	中路，打野	中路，下路	下路，补位	上路，中路
等待时间(s)	240	200	140	203	200
分配玩家	Player11	Player20	Player07	Player15	Player06
玩家偏好	上路，补位	中路，上路	中路，下路	打野，中路	辅助，中路
等待时间(s)	120	95	140	120	110

五、模型分析

纵观本题建模及模型求解的过程。我们首先将游戏位置分配问题抽象化，通过定义偏好向量、分配、可行域等符号及概念，将生活中的中的问题符号化、公式化。其次，为了评判一种分配方式的优劣性，又引入了损失函数的概念，将玩家满意度这一定性的描述转化为可以定量刻画的函数值。至此，我们为问题建立起了完备的数学结构，以此为依据寻找通用的求解方法加以分析解决。

对比爬山法、模拟退火算法、遗传算法三种模型求解方法，从最终解的损失函数值与结果直观分析上可以看出，遗传算法得到的解优于模拟退火算法，爬山法所得解的表现力最差。

爬山法存在一个较大的缺陷，简单地往低于当前解总成本的方向移动（沿着斜坡滑下），不一定会产生全局最优解，其所得到的解只能是一个局部范围内的最小值，难以实现全局最优。由于其巨大的缺陷，目前基本不被采用了，但其所包含的渐进优化思想、随机扰动思想、稳定迭代方向这几个核心要素，被后续随机优化算法所沿用。

在爬山法的基础上，模拟退火算法在迭代中不盲目地追求优解。其设定的温度、冷却速率、退火跃迁公式以及由这几个因素建立的决策方式，一定概率下允许在迭代过程中跃迁至较差的解，从而可能跳出局部优解，得到更优的解。

遗传算法的高效性离不开其具有的随机优化要素。通过变异和重组生成子代的过程，引入随机扰动，一定程度上缓解了损失函数陷入局部最优的问题。而模拟自然选择过程择优进入下一轮迭代，正是基于上一轮优化结果的进一步优化，此为渐进优化思想的体现。

六、模型总结

6.1 建模总结

多约束分配问题，一般表述为如何将有限的资源分配给多个表达了偏好的人，并尽可能使他们都满意，或尽可能使所有人都满意。常常体现为问题规模大、约束条件难以同时满足的特征。因此在解决这类问题上，要求在庞大的解空间中，以较少的运算次数、有意义的迭代方向，获得最大程度满足所有约束条件的最优解。可以看出，解决该问题，核心在三个要素：解空间的定义、解优劣性的判断方式、迭代优化算法。

解空间的定义

解空间$(domain)$，即所有有效解组成的集合，对于多约束分配问题，具体表现为所有可行分配方案。在本文研究的问题当中，我们将解空间定义为由游戏位置和玩家组成的元组

$(A\sim E/F, x_1\sim x_{20})$所组成的数组，且在一个解中同个玩家不能出现在不同位置。下式展示了其中一个可行解的示例。

$$[(A_0, x_{12}), (A_1, x_2), (B_0, x_{20}), (B_1, x_6), (C_0, x_3), (C_1, x_{16}), (D_0, x_9), (D_1, x_4), (E_0, x_{17}), (E_1, x_1)]$$

解优劣性的判断方式

解的优劣程度，通常使用该解的损失函数值来衡量，对于多约束分配问题，具体表现为该可行分配方案所违背的约束条件（总成本）。损失函数的设定，需要从预期达到的最优效果出发，考虑该分配问题的约束条件。大多情况下以多个约束因素作为变量建立函数，体现出当满足多个约束时函数值低、当满足较少约束时函数值高的特征。

迭代优化算法

采用有效的迭代算法，对求解的有效性和效率至关重要。多约束分配问题的求解方法较多，其中最基本的是遍历迭代计算，但是当问题规模大、解空间较大时，该算法需要过多的迭代次数和计算时间成本，效率低。本文对多约束分配问题的随机优化算法进行研究，主要有随机搜索法、爬山法、模拟退火算法、遗传算法。在第五章模型分析中可见其优劣性比较。

总结而言，使用随机优化算法解决多约束分配问题的一般步骤为：

1. 问题抽象。使用数学式子与变量抽象定义问题，定义解空间。
2. 定义损失函数。结合约束条件、预期最优效果建立损失函数。
3. 随机算法求解。选择合适的随机算法，调整相关参数，求解模型。

6.2 应用拓展

在生活中，多约束分配问题十分常见。就本文研究的问题而言，目前很多游戏都有玩家匹配环节，而支持游戏战场的服务器资源有限，玩家本身职能选择也有偏好，且在实际中我们也常常需要考虑玩家等待时间的长短，因此如何分配玩家，使得玩家职能尽可能与预期一致，且等待时间长的玩家尽可能优先进入游戏，这便是一个典型的多约束分配问题。

又例如处理带有偏好选择的学生宿舍安排问题。在选择宿舍时学生选择首选和次选宿舍，而宿舍数量和居住人数相对有限，则分配有限的宿舍需尽可能满足大多学生的需求，这就表现为多约束分配问题。诸如此类的还有，学生课程分配（学生对所选课程赋予权重）、家庭成员中的家务分配、企业的多预算资金分配等。

参考文献

[1] 姜启源，谢金星，叶俊.数学模型[M].第 5 版.北京：高等教育出版社，2018.5：1-18.
[2] 雷廷权，傅家骐.《热处理工艺方法 300 种 第二版》：机械工业出版社，1993.

附录

附录 1 损失函数核心代码(python)

```python
from const import *
from params import *
import math

def cost(pre, file, isLog):
    cost = 0.0
    pref_cost_value = 0.0
    time_cost_value = 0.0
    players_info = player_params.copy()

    # loop over each slot
    for i in range(len(pre)):
        playerIdx = int(pre[i])
        role_way = role_ways[slots[i]]
        this_player = players_info[playerIdx]
        this_player_name = this_player[0]
        this_pref = this_player[1]
        this_waitTime = this_player[2]

        # preference cost
        pref_cost_value += pref_cost(player=this_player_name, role_way=role_way, pref=this_pref)

        # waiting time cost
        time_cost_value += time_cost(wait_time=this_waitTime)

        cost = w_pref * pref_cost_value + w_time * time_cost_value

        # remove selected slot
        del players_info[playerIdx]

    if isLog == 1:
        # print to log.txt
        file.write('pref_cost: {:.4f}  time_cost: {:.4f}  total_cost: {:.4f}\n'.format(pref_cost_value, time_cost_value, cost))

    return cost

```

```
# player-player name, role_way-now way choice, pref-this player's preference
def pref_cost(player, role_way, pref):
    cost = 0

    # First pref costs 0, second choice costs 1, not on the list costs 3
    # While for ableAll player, first costs 0, other choice costs 2.5
    if player in ableAll:
        if pref[0] == role_way:
            cost += 0
        else:
            cost += 2.5
    else:
        if pref[0] == role_way:
            cost += 0
        elif pref[1] == role_way:
            cost += 1
        else:
            cost += 3

    return cost


def time_cost(wait_time):
    return exp_a/math.exp(wait_time/60)
```

附录 2 模拟退火算法核心代码(python)

```python
import random
import math

# Annealing Optimize
def annealingoptimize(domain, costf, file, T):

    times = 0
    minCost = 0.0

    # Initialize the values randomly
    now = [random.randint(domain[i][0], domain[i][1])
        for i in range(len(domain))]

    while T > 0.1:

        # log
        file.write('time {}:\n'.format(times))
        times += 1

        # Choose one of the indices
        i = random.randint(0, len(domain) - 1)

        # Choose a direction to change it
        dir = random.randint(-step, step)

        # Create a new list with one of the values changed
        next = now[:]
        next[i] += dir
        if next[i] < domain[i][0]:
            next[i] = domain[i][0]
        elif next[i] > domain[i][1]:
            next[i] = domain[i][1]

        # Calculate the current cost and the new cost
        ea = costf(now, file, 1)
        eb = costf(next, file, 0)

        minCost = ea

        # Is it better, or does it make the probability
        # cutoff?
        if (eb < ea):
            now = next
```

```
        else:
            p = pow(math.e, (-(eb - ea)) / T)
            if (p > random.random()):
                now = next

        # Decrease the temperature
        T = T * cool

        file.write('-------------------------------------------------------
    -----\n')

    return (now, minCost)
```

附录 3 遗传算法核心代码(Java)

```java
//GeneticOptimize.java
package Main;

import methods.Multiply;
import models.Distribution;
import models.Player;

import java.util.ArrayList;
import java.util.Comparator;

import static definition.Define.*;
import static methods.Init.initPlayer;
import static methods.Init.randomAlphaList;


public class GeneticOptimize {
    public static ArrayList<Player> players = initPlayer();     // 录入玩家信息

    public static void main(String[] args) {
        // 初始化种群
        ArrayList<Distribution> group = new ArrayList<>();
        for (int i=0; i<GROUP_NUM; i++) {
            group.add(new Distribution(randomAlphaList()));
        }
        // 种群迭代
        for (int i=0; i<ITERATION; i++) {
            // 按成本排序
            group.sort(Comparator.comparingDouble(d -> d.score));
            // 输出代最优
            System.out.println(i + " : " + String.format("%.2f", group.get(0).score));
            // 选择精英
            ArrayList<Distribution> elite = new ArrayList<>();
            int eliteNum = (int) (GROUP_NUM * ELITE_RATE);
            int cnt = 0;
            for (Distribution d : group) {
                elite.add(d);
                cnt++;
                if (cnt == eliteNum)
                    break;
            }
```

```
            // 繁衍
            group = Multiply.multiply(elite);
        }
        group.sort(Comparator.comparingDouble(d -> d.score));
        System.out.println("Best performance : " + group.get(0).code + " ---- " + String.format("%.2f", group.get(0).score));
    }
}
```

```
//Multiply.java
package methods;

import models.Distribution;

import java.util.ArrayList;

import static definition.Define.*;
import static methods.Calculate.calculateScore;

public class Multiply {
    public static ArrayList<Distribution> multiply(ArrayList<Distribution> elite) {
        // 基因突变
        ArrayList<Distribution> mutateGroup = new ArrayList<>();
        for (Distribution d : elite) {
            if (Math.random() < MUTATE_RATE) {
                // 从 candidate 突变
                int i1 = (int) (Math.random()*10);
                int i2 = (int) (Math.random()*20);
                char i1c = d.code.charAt(i1);
                char i2c = (char) (i2 + 'a');
                while (d.code.contains(Character.toString(i2c))) {
                    i2 = (int) (Math.random()*20);
                    i2c = ((char) (i2 + 'a'));
                }
                String str = d.code.replace(i1c, i2c);
                mutateGroup.add(new Distribution(str));
            } else {
                mutateGroup.add(new Distribution(d.code));
            }
        }

        // 基因重组
        ArrayList<Distribution> newGroup = new ArrayList<>();
        for (int i=0; i<GROUP_NUM/2; i++) {
```

```
36.             int i1 = (int) (Math.random()*GROUP_NUM*ELITE_RATE);
37.             int i2 = (int) (Math.random()*GROUP_NUM*ELITE_RATE);
38.             while (i1 == i2) i2 = (int) (Math.random()*GROUP_NUM*ELITE_RATE)
    ;
39.             String str1 = mutateGroup.get(i1).code;
40.             String str2 = mutateGroup.get(i2).code;
41.             char[] charList1 = str1.toCharArray();
42.             char[] charList2 = str2.toCharArray();
43.             // 一次结合
44.             for(int j=0; j<10; j++) {
45.                 if (Math.random() < RECOMBINE_RATE) {
46.                     if (!str2.contains(Character.toString(charList1[j])) &&
    !str1.contains(Character.toString(charList2[j]))) {
47.                         str1 = str1.replace(Character.toString(charList1[j])
    , Character.toString(charList2[j]));
48.                         str2 = str2.replace(Character.toString(charList2[j])
    , Character.toString(charList1[j]));
49.                     }
50.                 }
51.             }
52.             newGroup.add(new Distribution(str1));
53.             newGroup.add(new Distribution(str2));
54.         }
55.
56.         return newGroup;
57.
58.     }
59. }
```