Genetic-algorithm

胡家田

遗传算法概论

遗传算法定义：

遗传算法（Genetic Algorithm，简称GA）起源于对生物系统所进行的计算机模拟研究，是一种**随机全局搜索优化**方法，它模拟了自然选择和遗传中发生的**复制**、**交叉**(crossover)和变异(mutation)等现象，从任一初始种群（graph_dict）出发，通过随机选择、交叉和变异操作，产生一群更适合环境的个体，使群体进化到搜索空间中越来越好的区域，这样一代一代不断繁衍进化，最后收敛到一群最适应环境的个体（Individual），从而求得问题的优质解。

遗传算法适用范围：

1. **复杂的优化问题**：遗传算法通常在复杂的优化问题中表现出色，其中存在大量的搜索空间和约束条件。这包括**组合优化、连续优化、离散优化**等各种类型的问题。
2. **无法解析求解的问题**：对于那些没有明确的数学解析解的问题，遗传算法是一种有效的求解工具，因为它们**不依赖于问题的解析性质。**
3. **全局搜索**：遗传算法是全局搜索算法，能够搜索整个解空间，以寻找可能的**全局最优解**，而不容易陷入局部最优解。
4. **多模态问题**：在多模态问题中，解空间包含多个局部最优解。遗传算法能够探索不同的解，并在**多个局部最优解之间进行切换。**
5. **参数优化**：在机器学习、深度学习和模型调优中，遗传算法可以用于**优化模型参数，以改善模型性能。**
6. **进化和遗传系统模拟**：遗传算法模拟生物进化过程，因此它们在研究**生物学、生态学和进化系统建模**中有应用。
7. **路径规划和调度**：遗传算法可用于解决路径规划、旅行商问题、车辆路径问题等**调度和路线优化**问题。
8. **特征选择和维度规约**：在数据挖掘和机器学习中，遗传算法可用于**选择最相关的特征或降低数据维度。**
9. **仿真和模拟**：在复杂系统建模、天气预测、交通流量模拟等领域，遗传算法用于**生成模型参数或优化仿真模型。**
10. **机器学习超参数优化**：对于机器学习算法中的超参数调整，遗传算法可以帮助**寻找最佳超参数组合。**

遗传算法的基本单位概念

① 染色体(Chromosome)：染色体又可称为基因型个体(individuals)，一定数量的个体组成了群体(graph_dict)，群体中个体的数量叫做群体大小（graph_dict size）。 **（染色体是进行选择、遗传和变异的基本单位）**

② 位串(Bit String)：个体的表示形式。对应于遗传学中的染色体。

③ 基因(Gene)：基因是染色体中的元素，用于表示个体的特征。例如有一个串（即染色体）S=1011，则其中的1，0，1，1这4个元素分别称为基因。 **（基因可以是解空间构成最优解的元素映射集）**

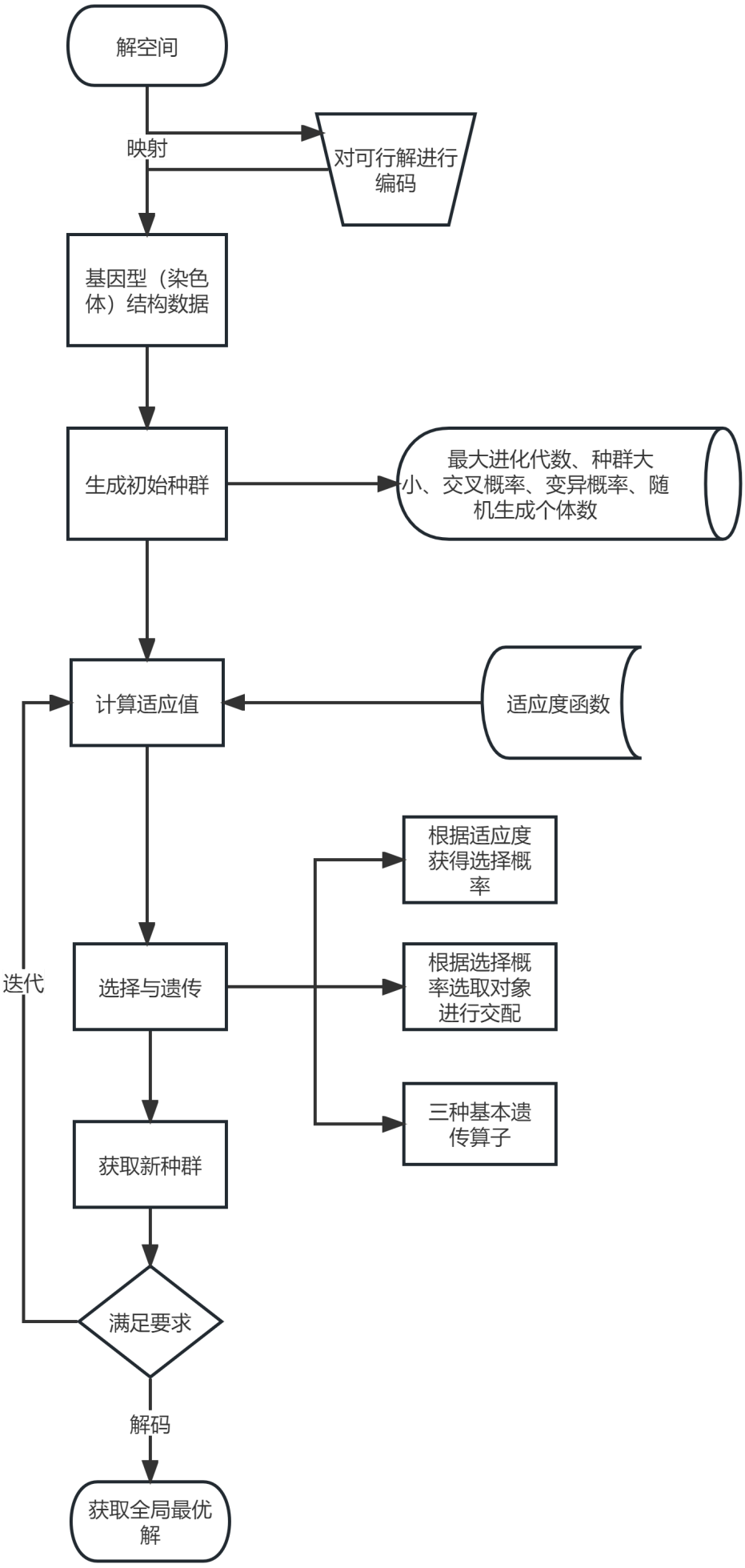
④ **特征值(Feature)：**在用串表示整数时，基因的特征值与二进制数的权一致；例如在串 S=1011 中，基因位置3中的1，它的基因特征值为2；基因位置1中的1，它的基因特征值为8。（这取决于编码方式）

⑤ **适应度(Fitness)：**各个个体对环境的适应程度叫做适应度(fitness)。为了体现染色体的适应能力，引入了对问题中的每一个染色体都能进行度量的函数，叫适应度函数。这个函数通常会被用来计算个体在群体中被使用的概率。**(适应度函数应当能通过染色体的编码方式反映出其接近最优解的程度，这是至关重要的)（适应度函数应当反映解码函数的一部分）**

⑥ **表现型(Phenotype)：**生物体的基因型在特定环境下的表现特征。对应于GA中的位串解码后的参数。 **（也就是解空间到基因型的双向映射）**

遗传算法基本过程

基本遗传算法流程图:


解空间
映射
对可行解进行编码
基因型（染色体）结构数据
生成初始种群
最大进化代数、种群大小、交叉概率、变异概率、随机生成个体数
计算适应值
适应度函数
根据适应度获得选择概率
选择与遗传
根据选择概率选取对象进行交配
三种基本遗传算子
迭代
获取新种群
满足要求
解码
获取全局最优解

遗传算法的关键步骤

染色体编码：

染色体编码是把问题的解空间映射到基因型的过程，这里由基因单位本身的离散队列和对队列元素选择不同编码方式的两部分构成，给予了映射处理较大的操作空间，映射时应当考虑后续适应度函数的设计和自然选择部分（**可以说映射设计和适应度函数与遗传设计直接决定了算法优化性能，且前者很大程度上影响后者的设计**）考虑如下情形：

- 解空间可以分解为（指更高的抽象过程）离散型的基本单位，直接实现基本单位与基因单位的一一映射，再对基因单位进行编码（其实此处编码没有太大必要，直接使用有序队列）。
- 解空间本身就是离散型的单位或者可离散的对象，通过编码得到的离散型队列实现与解空间对应。

常用编码方式：

1. **二进制编码**：二进制编码是将问题的解表示为**二进制字符串**的方式。每个属性或参数通常由一组二进制位表示。这是遗传算法中最常见的编码方式，适用于许多问题，包括优化问题、参数优化等。
2. **整数编码**：在整数编码中，问题的解被表示为**整数值**。这适用于那些属性或参数是整数的问题，例如排课问题或车辆路径问题。
3. **实数编码**：实数编码用于表示问题的解为实数值。这种编码方式适用于**连续优化问题**，如**函数优化或参数调整**。
4. **排列编码**：排列编码用于处理排列问题，例如旅行商问题（TSP），其中解被表示为**元素的排列**。每个元素在排列中只出现一次。
5. **集合编码**：在集合编码中，问题的解表示为**一组元素的集合**。这种编码方式适用于**集合覆盖问题、子集选择**等问题。
6. **树编码**：树编码用于解决树结构相关的问题，例如**表达式树、决策树、或树形组织的结构**。
7. **图编码**：图编码适用于问题中的图结构，如**图的着色问题、图的路径问题**等。
8. **字符串编码**：字符串编码将问题的解表示为字符串。这适用于某些**文本生成问题、密码破解**等。
9. **问题特定编码**：某些问题可能需要特定的编码方式，根据**问题的性质来设计自定义编码方式**。这种方式通常用于特定领域的问题。
10. **混合编码**：在某些情况下，问题的解可能包含多个部分，**每个部分可以使用不同的编码方式**。这称为混合编码，用于**复杂的多模态问题**。

可以考虑将解空间先进行一步抽象（如抽象成数据结构），以便观察解空间的构成结构，从而进行后续的映射设计。当然，面对现实情况还是需要切合实际的处理（如混合编码、自定义编码等待），以整体设计架构为优为主要目的。

考虑编码中的对应过程，未对应前基因单位是无意义的0和1的组合，而解空间的基本单位都是有特定属性的组合，所以构建个体的属性赋值也是一个需要考虑的点：

- 随机初始化
- 启发式初始化：使用问题**特定的启发信息**来初始化个体的属性
- 有**约束**的初始化
- 基于问题的**特定**初始化
- 自适应方法：属性值的构建或修改受到算法**自身的控制和调整**

但注意，要保证种群的**差异性**，防止变异率不足。

设计适应度函数：

适应度函数的根本要求：**反映个体的优劣**（即能通过染色体的编码方式反映出其接近最优解的程度）（这是自然选择的核心，也是实现优化的核心）

适应度尺度变换：指算法迭代的不同阶段，能够通过适当改变个体的适应度大小，进而避免群体间适应度相当而造成的竞争减弱，导致种群收敛于局部最优解。

尺度变换选用的经典方法：线性尺度变换、乘幂尺度变换以及指数尺度变换。

设计选择和交叉：

选择：**选择操作从旧群体中以一定概率选择优良个体组成新的种群，以繁殖得到下一代个体。**个体被选中的概率跟适应度值有关，个体适应度值越高，被选中的概率越大。

以赌轮盘法为例：

$$P_i = \frac{f_i}{\sum_{k=1}^{M} f_k}$$

常用选择方法：

1. **轮盘赌选择**（Roulette Wheel Selection）：轮盘赌选择是一种基于概率的选择方法，个体被选中的概率与其适应度值成正比。**适应度值高的个体更有可能被选中**，但并不保证最佳个体会被选中。
2. **锦标赛选择**（Tournament Selection）：锦标赛选择是通过随机选择若干个个体，然后从中选择适应度最高的个体。这个方法相对简单，适合**快速选择个体。**
3. **最佳个体选择**（Elitism）：在选择中，**保留种群中适应度最高的个体**，确保这些个体会传递到下一代。这有助于保留最佳解决方案。
4. **排序选择**（Rank Selection）：在排序选择中，种群中的个体根据其适应度值的排名来选择。通常，**排名高的个体更容易被选中。**
5. **随机选择**（Random Selection）：在随机选择中，个体被纯随机选择，**不考虑其适应度值。**这种选择方法通常不会引导搜索，但在某些情况下可能有用。
6. **指数选择**（Exponential Selection）：指数选择是基于个体的适应度值来确定选择概率，但它使用指数函数来增强选择压力，以确保**较差的个体也有机会被选中。**
7. **拥挤度选择**（Crowding Selection）：拥挤度选择是**针对多目标优化问题设计**的，它考虑了个体的密度，以确保较少探索的区域得到适当的关注。
8. **淘汰选择**（Steady-State Selection）：在淘汰选择中，每一代只选择少数个体，并将它们替换到种群中。这种方法通常与遗传算法的**演化策略**一起使用。
9. **自适应选择**（Adaptive Selection）：在自适应选择中，选择概率会**根据种群的性能和搜索进展动态调整。**这有助于改进选择策略以适应问题的不确定性。

交叉：从种群(选择生成的新种群）中随机选择（此处的选择还可以再考虑）（也可以在选择的同时进行交叉遗传，对结束后的总种群进行优胜劣汰或者其它选择）两个个体，通过两个染色体的交换组合，把父串的优秀特征遗传给子串，从而产生新的优秀个体。其重点在于遗传过程，**交叉过程能否实现将父代中倾向于使结果偏向最优解的基因单位遗传给下一代，或者使遗传结果的基因单位的有序队列的总体表现优于父代是至关重要的。**

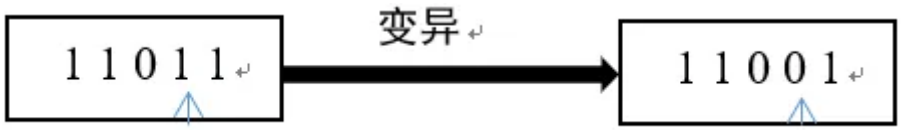
常用的交叉方法有：

1. **一点交叉**（One-Point Crossover）：在一点交叉中，**随机选择一个交叉点**，然后将两个父代的染色体在这个交叉点处切割，然后交换各自的部分，以生成两个子代。这是一种常见的二进制编码和整数编码的交叉方法。

2. **两点交叉**（Two-Point Crossover）：与一点交叉类似，但在**两个交叉点处切割染色体**，然后交换中间部分，以生成两个子代。
3. **均匀交叉**（Uniform Crossover）：在均匀交叉中，对**每个基因位（比特或属性）进行随机选择**，然后交换两个父代对应位置上的基因。这种交叉方法通常用于二进制编码。
4. **多点交叉**（Multi-Point Crossover）：在多点交叉中，**多个交叉点被选择**，并且父代染色体在这些点处切割并交换，以生成子代。
5. **顺序交叉**（Order Crossover，OX）：顺序交叉主要**用于排列编码问题**，如TSP。在顺序交叉中，首先随机选择一个子串，然后将该子串的顺序保留，然后按照另一个父代的顺序填充剩余部分。
6. **部分映射交叉**（Partially Mapped Crossover，PMX）：部分映射交叉也**用于排列编码问题**，它首先选择两个交叉点，然后交换这两个交叉点之间的基因，同时保留其他部分的顺序。
7. **环形交叉**（Cycle Crossover）：环形交叉也**用于排列编码问题。**它从一个父代染色体开始，并从第一个交叉点的位置开始创建一个循环，然后从另一个父代染色体中填充未被选择的元素。
8. **部分匹配交叉**（Partially Matched Crossover，PMXC）：部分匹配交叉也用于排列编码问题，它尝试**保持父代染色体之间的局部相似性。**
9. **边交叉**（Edge Recombination Crossover，ERX）：边交叉**用于TSP问题**，它尝试保留两个父代染色体中的边信息，以生成子代。
10. **线性交叉**（Linear Crossover）：线性交叉是一种适用于**实数编码问题**的方法，它在两个父代之间**生成线性关系**以生成子代。

设计变异：

为了**防止遗传算法在优化过程中陷入局部最优解**，在搜索过程中，需要对个体进行变异，**在实际应用中，主要采用单点变异，也叫位变异，即只需要对基因序列中某一个位进行变异，以二进制编码为例，即0变为1，而1变为0。**



群体P（t）经过选择、交叉、变异运算后得到下一代群体 P(P+1) 。

变异的设计有以下考虑：

1. **变异率的选择**：确定适当的变异率是关键。如果变异率太低，可能不足以引入足够的多样性，而如果变异率太高，可能会导致收敛速度较慢或算法失效。通常，可以进行实验来确定最佳的变异率。
2. **变异操作的类型**：不同的问题和编码方式可能需要不同类型的变异操作。一些常见的变异操作类型包括**随机变异、位变异、数值变异、交换变异**等。选择适当的变异操作类型与问题的性质和编码方式有关。
3. **变异范围的设置**：如果问题的解空间有约束条件，需要**确保变异操作不会生成无效的解。**在设计变异操作时，需要考虑解的合法性。
4. **自适应变异**：一些遗传算法采用**自适应策略**，其中变异率会根据算法的性能和搜索进展动态调整。这有助于在搜索过程中平衡多样性和收敛性。
5. **局部变异**：在遗传算法的后续迭代中，可以对个体进行**局部变异**，以**微调其属性值。**这有助于进一步改善解的质量。
6. **变异算子的选择**：可以考虑**使用多种不同的变异算子**，每种变异算子用于不同的情况。通过以一定概率选择不同的变异算子，可以增加算法的多样性。
7. **问题特定的变异**：某些问题可能具有特定的特性，需要**设计问题特定的变异操作。**例如，在图像生成问题中，可以设计特定的图像操作来进行变异。
8. **多层次变异**：可以考虑**多层次的变异**，其中不同层次的变异操作引入不同程度的变化。例如，全局变异和局部变异可以结合使用。

9. **变异操作的控制**：可以使用自适应控制方法来**动态调整变异操作的参数**，例如，变异幅度或变异步长。
10. **实验和调整**：设计变异操作通常需要进行**实验和参数调整。**通过观察算法的性能并根据实验结果调整变异操作，可以改进算法的搜索能力

设计终止判断条件：

若 $t \leq T$ ，则 $t \leftarrow t+1$ ，进行迭代；否则以进化过程中所得到的**具有最大适应度的个体作为最好的解输出**，终止运算。

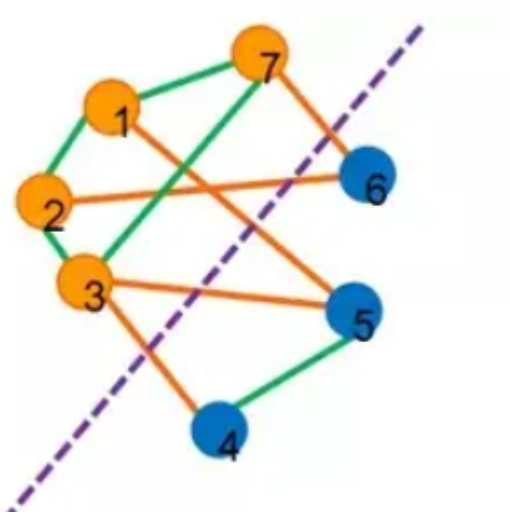
有以下设计方法：

1. **固定迭代次数**：最简单的终止条件之一是指定算法运行的固定迭代次数。这意味着算法将运行固定的代数，不管是否已找到满意的解决方案。这是一种**简单且可控**的方式，但不适用于所有问题，因为不同问题的优化难度不同。
2. **最优解稳定性**：可以**检查每一代中最优解的适应度值**，如果在一定代数内最优解没有显著改进，算法可以终止。这表示算法已经收敛到局部最优解，并且不太可能再找到更好的解。
3. **适应度值收敛**：可以监测种群的**平均适应度值**，如果平均适应度值在一定代数内趋于稳定，算法可以终止。这表明种群已经趋于稳定，不太可能有更多的进展。
4. **满足问题特定条件**：有时，问题**本身具有终止条件。**例如，在路径规划问题中，可以设置终止条件为达到目标位置或满足特定约束条件。
5. **时间限制**：设置算法运行的最大**时间限制**，一旦达到这个时间限制，算法将终止。这适用于需要在有限时间内执行的任务。
6. **适应度阈值**：指定一个**适应度阈值**，一旦找到适应度值达到或超过该阈值的解，算法可以终止。
7. **种群多样性**：监测种群的**多样性**，如果多代之后种群变得不太多样，算法可能陷入局部最优解，可以考虑终止算法并重新初始化种群。
8. **自适应终止**：采用**自适应策略**，根据算法的性能和搜索进展来调整终止条件。例如，可以根据搜索进展动态调整迭代次数或时间限制。

基本遗传算法的代码实现

基本编程思维：面向过程编程

以Max-cut problem问题为例，即将一个无向图切成2个部分（子图），从而使得2个子图之间的边数最多。



1.染色体编码：

解空间是不同切割方式得到的子图联系边数（示意图为5）的离散序列，考虑到此序列可以结构为更基本的组成单位——不同的结点，以结点的有序队列与基因单位进行映射，并采用二进制编码方式，七个位置指定赋予1~7的结点。0为一种子图，1为另一种子图。

2.初始参数设定和初始种群生成：

```
graph_dict_Size:10

Crossover_Probability:0.8

Mutation_Probability:0.05

graph_dict = {}

# 遍历随机生成10个个体 也可以使用贪心算法生成 但要先写适应度函数
for i in range(1, 11):
    key = f'graph{i}'
    value = ''.join(random.choice('01') for _ in range(7))
    graph_dict[key] = value
```

3.赌轮盘法选择两个父代：

```
def select_Polpulation(dictionary):
    keys = list(dictionary.keys())

    # 从字典中随机选择两个键
    selected_keys = random.sample(keys, 2)

    # 获取选中键对应的值
    selected_values = [dictionary[key] for key in selected_keys]

    return selected_values
```

4.进行杂交并生成子代：

```
def compare_and_replace(strings):

    for i in range(len(strings[0])):
        if strings[0][i] == strings[1][i]:
            result += strings[0][i]
        else:
            result += random.choice("01")

    return result
```

5.子代进行变异：

```
def mutate_bit(input_string, mutation_rate=0.05):
    if not (0 <= mutation_rate <= 1):
        raise ValueError("Mutation rate should be between 0 and 1.")

    mutated_string = ""
    for bit in input_string:
        if random.random() < mutation_rate:
            # 随机决定是否变异，以mutation_rate的概率
            mutated_bit = "0" if bit == "1" else "1"
            mutated_string += mutated_bit
        else:
            mutated_string += bit

    return mutated_string
```

6.设计适应度函数：

```
def classify_nodes(graph_str):
    class1 = []  # 存储分类为1的节点
    class2 = []  # 存储分类为2的节点
    edges = []   # 存储图的边关系

    # 根据输入字符串构建节点分类和边关系
    for i, char in enumerate(graph_str):
        if char == '0':
            class1.append(i + 1)  # 节点编号从1开始
        else:
            class2.append(i + 1)

    # 定义图的连接关系
    graph = {
        1: [2, 5, 7],
        2: [3, 6,1],
        3: [4, 7,5],
        4: [5,3],
        5: [1,4,3],
        6: [7,2],
        7: [1,3]
    }

    # 构建图的边关系
    for node, neighbors in graph.items():
        for neighbor in neighbors:
            if node < neighbor:
```

```
                edges.append((node, neighbor))

    # 寻找属于不同类别的节点之间的连接关系
    connected_pairs = 0
    for edge in edges:
        if (edge[0] in class1 and edge[1] in class2) or (edge[0] in class2 and edge[1] in
class1):
            connected_pairs += 1

    return connected_pairs
```

7.选择并产生新种群：

```
def replace_lowest_value_with_graph(input_string, graph_dict):
    # 初始化最低值和对应的键
    lowest_value = float('inf')
    lowest_key = None

    # 遍历字典，计算各值并找到最低值的键
    for key, value in graph_dict.items():
        result = classify_nodes(value)
        if result < lowest_value:
            lowest_value = result
            lowest_key = key

    # 替换最低值的键对应的值为输入的01字符串
    graph_dict[lowest_key] = input_string

    return graph_dict
```

8.设计解码函数：

```
def find_max_classification_result(graph_dict):
    max_result = float('-inf')  # 初始化最大结果为负无穷

    for value in graph_dict.values():
        result = classify_nodes(value)
        max_result = max(max_result, result)

    return max_result
```

9.设计主函数并开始迭代：

```
def main():
    graph_dict = {}

# 遍历随机生成10个个体 也可以使用贪心算法生成 但要先写适应度函数
    for i in range(1, 11):
        key = f'graph{i}'
        value = ''.join(random.choice('01') for _ in range(7))
        graph_dict[key] = value

    num_iterations = 1000

    for _ in range(num_iterations):
        selected_values=select_Polpulation(graph_dict)
```

```
        result=compare_and_replace(selected_values)
        mutated_string=mutate_bit(result, mutation_rate=0.05)
        graph_dict=replace_lowest_value_with_graph(mutated_string, graph_dict)

    max_result=find_max_classification_result(graph_dict)
    print(max_result)

if __name__ == "__main__":
    main()
```

10.输出结果：

```
[Running] python -u "e:\Git-Repositories\Mathematical_modeling\Mathematical-Modeling_Knowledge-reserve\Algorithm\Modern_Optimization-algorithms\Genetic_Test.py"
9
```