八皇后问题的局部搜索算法实验报告

钟赟 202028013229148

一、问题背景和定义

八皇后问题（Eight queens）是国际象棋棋手马克斯·贝瑟尔于1848年提出的问题。

问题描述为：在8×8格的国际象棋上摆放8个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。求解一种可行的皇后摆法。

我们定义变量 $CONFIG = \{config_0, config_1, \ldots, config_7\}$ 来记录皇后在棋盘上的位置，其中 $config_i$ 表示第 i 列的皇后所在的行号。

二、迭代改进法

（一）算法设计

迭代改进法是最基础的局部搜索策略，它本质上是一种贪心的策略。首先在搜索空间中随机选择一点作为初始解，在搜索的每一步，都会从当前节点的邻域中选择一个评估函数更优的解作为当前的最优解，直到邻域中不存在满足条件的解为止。

首先将 POS 初始化为随机变量。棋盘上除去皇后所占的 8 个格点之外剩余 56 个空格均作为解的邻域，评估函数为当前棋盘上互相冲突的皇后对数。在邻域中迭代搜索，直到无法找到评估函数更小的更优解为止。

（二）编程实现

```
import random

# 评估函数，计算给定格局下的冲突皇后对数
def eval(config):
    cnt = 0
    for col0 in range(0,7):
        for col1 in range(col0+1,8):
            if(config[col0] == config[col1]) or abs(config[col0] - config[col1])
== abs(col0-col1):
                cnt += 1
    return cnt

# 迭代改进算法：寻找邻域内的最优解，若不存在，则返回原来的格局
def Iteration_Optimize(config):
    best_config=config
    for col in range(0,8):
        for row in range(0,8):
            if config[col]==row :
                continue
            new_config=config[:]
            new_config[col]=row
            if eval(new_config) < eval(best_config):
                best_config=new_config
    return best_config
```

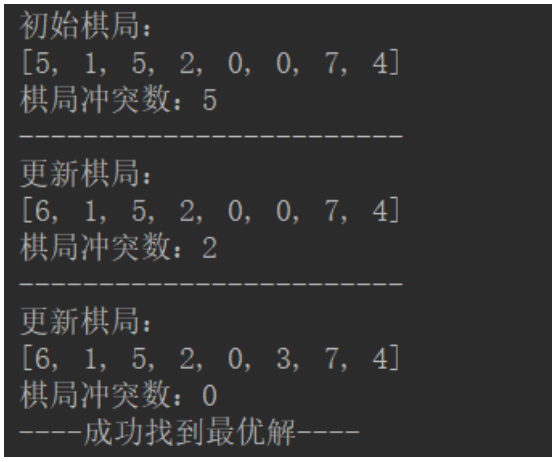
```
##############################################
# 定义皇后的位置，数组每一位代表该列的皇后所在的行数
config = [0,0,0,0,0,0,0,0]
# 生成初始随机解
for col in range(0,8):
    config[col]=random.randint(0,7)

print("初始棋局：")
print(config)
print("棋局冲突数："+str(eval(config)))

# 迭代循环直到找到冲突数为0
while eval(config) > 0:
    new_config=Iteration_Optimize(config)
    # 迭代找到的最优邻居和自己相同，说明无法找到更优的邻居，退出循环
    print("------------------------")
    print("更新棋局：")
    print(new_config)
    print("棋局冲突数："+str(eval(new_config)))
    if new_config == config:
        print("----寻找最优解失败----")
        break
    if eval(new_config) == 0:
        print("----成功找到最优解----")
    config=new_config
```

（三）实验结果

失败结果如下：

```
初始棋局：
[6, 7, 5, 6, 5, 2, 4, 6]
棋局冲突数：8
------------------------
更新棋局：
[0, 7, 5, 6, 5, 2, 4, 6]
棋局冲突数：5
------------------------
更新棋局：
[0, 7, 1, 6, 5, 2, 4, 6]
棋局冲突数：3
------------------------
更新棋局：
[0, 7, 1, 3, 5, 2, 4, 6]
棋局冲突数：2
------------------------
更新棋局：
[0, 7, 1, 3, 5, 2, 4, 6]
棋局冲突数：2
----寻找最优解失败----
```

成功结果如下：



实验过程中，只有少部分实验能够求得最优解，绝大部分棋局是无法找到最优解的，说明迭代改进法的效率较低。

三、模拟退火法

（一）算法设计

迭代改进算法很容易陷入局部最优解，对于初始解的选择要求比较高，模拟退火算法可以对其进行改进。模拟退火算法来源于冶金学，引入一个介于 0 和 1 之间的参数 ϵ，如果有比较好的解则接受，否则会以 ϵ 的概率接受比较差的解，类似于多次重启，避免陷入局部最优解。

在八皇后问题中，随机选取 56 个邻居中的一个，如果得到的新解更优，则选择该邻居，如果更差，则以概率 ϵ 选择该邻居状态。其中，ϵ 逐渐减少直到接近 0，当 ϵ 下降得足够慢，找到全局最优解的概率就接近 1。

（二）编程实现

```
import random
import math

# 评估函数，计算给定格局下的冲突皇后对数
def eval(config):
    cnt = 0
    for col0 in range(0,7):
        for col1 in range(col0+1,8):
            if(config[col0] == config[col1]) or abs(config[col0] - config[col1])
== abs(col0-col1):
                cnt += 1
    return cnt

# 模拟退火算法：根据温度T和当前格局返回一个新格局
def Simulate_Annealing(config,T):
    # 用 neighbor 记录当前棋局的所有邻居棋局
    neighbor=[]
    for col in range(0,8):
        for row in range(0,8):
            new_config=config[:]
            if config[col]!=row :
                new_config[col]=row
                neighbor.append(new_config)
    # 从邻居中随机选取一个棋局
    picked=random.randint(0,55)
    if eval(neighbor[picked]) < eval(config):# 随机选择的棋局比原棋局更优
        return neighbor[picked]
```

```
    else:# 随机选择的棋局比原棋局更差，以概率p选择它
        p=math.e**((eval(config)-eval(neighbor[picked]))/T)
        if p > random.randint(0,1000)/1000:
            return neighbor[picked]
    return config

##############################################
# 定义皇后的位置，数组每一位代表该列的皇后所在的行数
config = [0,0,0,0,0,0,0,0]
# 定义初始温度，在迭代过程中递减
T = 5.0

# 生成初始随机解
for col in range(0,8):
    config[col]=random.randint(0,7)

print("初始棋局：")
print(config)
print("棋局冲突数："+str(eval(config)))

# 迭代循环直到温度T为0
while T > 0.0001:
    while eval(config) > 0:
        print("-------------------------")
        new_config=Simulate_Annealing(config,T)
        # 迭代找到的最优邻居和自己相同，说明此步没有优化
        if new_config == config:
            print("----没有移动----")
            break
        print("更新棋局：")
        print(new_config)
        print("棋局冲突数："+str(eval(new_config)))
        if eval(new_config) == 0:
            print("----成功找到最优解----")
        config=new_config
    T = T * 0.99

if T <= 0.0001 and eval(new_config) > 0:
    print("----寻找最优解失败----")
```

（三）实验结果

失败结果如下：

```
初始棋局：
[6, 0, 0, 3, 3, 4, 3, 7]
棋局冲突数：10
------------------------
更新棋局：
[4, 0, 0, 3, 3, 4, 3, 7]
棋局冲突数：10
------------------------
----没有移动----
------------------------
更新棋局：
[4, 6, 0, 3, 3, 4, 3, 7]
棋局冲突数：8
------------------------
更新棋局：
[4, 6, 0, 3, 7, 4, 3, 7]
棋局冲突数：5
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
更新棋局：
[4, 6, 0, 6, 7, 4, 3, 7]
棋局冲突数：7
------------------------
更新棋局：
[4, 6, 0, 6, 7, 4, 3, 3]
棋局冲突数：7
------------------------
更新棋局：
[4, 6, 2, 6, 7, 4, 3, 3]
棋局冲突数：8
```

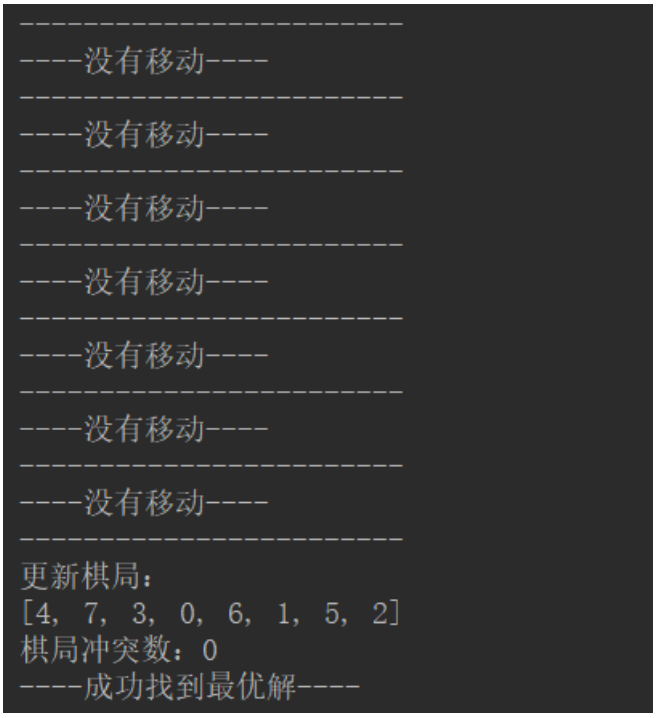
.....省略中间迭代过程

```
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
----没有移动----
------------------------
更新棋局：
[0, 5, 3, 1, 7, 2, 2, 6]
棋局冲突数：1
------------------------
----没有移动----
------------------------
----没有移动----
----寻找最优解失败----
```

成功结果如下：

```
初始棋局：
[4, 4, 4, 7, 4, 2, 4, 6]
棋局冲突数：12
------------------------
----没有移动----
------------------------
更新棋局：
[4, 4, 7, 7, 4, 2, 4, 6]
棋局冲突数：9
------------------------
更新棋局：
[5, 4, 7, 7, 4, 2, 4, 6]
棋局冲突数：7
------------------------
更新棋局：
[5, 4, 7, 5, 4, 2, 4, 6]
棋局冲突数：7
------------------------
更新棋局：
[5, 4, 7, 5, 4, 2, 4, 2]
棋局冲突数：9
------------------------
更新棋局：
[4, 4, 7, 5, 4, 2, 4, 2]
棋局冲突数：9
------------------------
更新棋局：
[4, 4, 7, 5, 4, 6, 4, 2]
棋局冲突数：8
------------------------
更新棋局：
[4, 4, 7, 5, 4, 3, 4, 2]
棋局冲突数：11
------------------------
```

.....省略中间迭代过程



从实验结果可以看出，随着迭代次数增加，随机选择的新棋局与原棋局相同的概率越来越大，后期的收敛速度变慢。但寻找最优解的效率明显优于迭代改进法。

四、遗传算法

（一）算法设计

遗传算法来源于种群的进化，遵循优胜劣汰、适者生存。遗传算法中，选择、交配、变异是最主要的三个操作：根据适应值的大小，选择可以从规模为 M 的群体中选择若干染色体构成种群，种群可以与群体规模一致，也可以不一致，且适应性更好的染色体更有可能在种群中存在多个，而那些适应值比较小的染色体有可能就会被淘汰，后面均假设选出来的种群个数为N。交配指的是从种群中随机的选取两个染色体，将染色体的某段基因互换。

在八皇后问题中，随机生成 K 个解，以一定概率选择两个解，通过交换后面 N（0 < N < 8）个皇后的位置获得两个后代，同时将 K 个后代继续配对，配对过程中以 ϵ 的概率产生变异。实验中，设置 K = 4, N = , ϵ = 。

（二）编程实现

```
import random

# 评估函数，计算给定格局下的不冲突皇后对数
def eval(config):
    cnt = 0
    for col0 in range(0,7):
        for col1 in range(col0+1,8):
            if(config[col0] == config[col1]) or abs(config[col0] - config[col1])
== abs(col0-col1):
                cnt += 1
    return 28-cnt

# 从种族中选择邻域中的某个棋局作为父代，冲突对数越少的棋局被选中的概率更大
def get_parent(family,conflict):
    choose=random.randint(0,sum(conflict)-1)
    if choose < conflict[0]:
        return family[0]
```

```
    elif choose>=conflict[0] and choose<conflict[0]+conflict[1]:
        return family[1]
    elif choose>=conflict[0]+conflict[1] and
choose<conflict[0]+conflict[1]+conflict[2]:
        return family[2]
    else:
        return family[3]

# 变异过程
def variation(family):
    for ind in range(0,K):
        col=random.randint(0,7)
        row=random.randint(0,7)
        family[ind][col]=row
    return family

# 杂交 种族中两两进行杂交
def crossbreed(family):
    conflict=[]
    new_family=[]
    for ind in range(0,K):
        conflict.append(eval(family[ind]))
    for i in range(0,K-2):
        first_child=family[i*2][:]
        second_child=family[i*2+1][:]
        # 交换下标为 0-num 的皇后，完成杂交
        num=random.randint(0,6)
        for j in range(0,num+1):
            first_child[j]=second_child[j]
            second_child[j]=family[i*2][j]
        new_family.append(first_child)
        new_family.append(second_child)
    return  new_family

# 判断该种族是否有可行解
def find_answer(family):
    for ind in range(0,K):
        # print(eval(family[ind]))
        if eval(family[ind])==28:
            print("----成功找到最优解----")
            print(family[ind])
            return True
    return False

##############################################
# 定义种族大小
K = 4
# 随机生成K个棋局，作为种族
family = []
for col in range(0,K):
    config = [0,0,0,0,0,0,0,0]
    for col in range(0,8):
        row=random.randint(0,7)
        config[col]=row
    family.append(config)
print("初始棋局种族：")
print(family)
```

```
# 种族进行杂交
family=crossbreed(family)

# 迭代循环直到找到最优后代
while find_answer(family)==False:
    print("------------------------")
    # 10%的概率进行变异
    p=random.randint(1,10)
    if p==1:
        print("进行变异，变异后的种族为：")
        family=variation(family)
        print(family)
    else: # 进行杂交
        family=crossbreed(family)
        print("进行杂交，杂交后的种族为：")
        print(family)
```

（三）实验结果

由于存在变异，只要迭代下去，遗传算法总能找到可行解，示例结果如下：

```
------------------------
进行杂交，杂交后的种族为：
[[6, 6, 7, 6, 3, 7, 5, 1], [6, 1, 2, 3, 7, 7, 6, 3], [1, 7, 5, 5, 5, 2, 3, 4], [0, 7, 6, 2, 5, 7, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 1, 2, 3, 7, 7, 6, 1], [6, 6, 7, 6, 3, 7, 5, 3], [0, 7, 6, 2, 5, 2, 3, 4], [1, 7, 5, 5, 5, 7, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 1, 2, 3, 7, 7, 6, 1], [6, 6, 7, 6, 3, 7, 5, 3], [1, 7, 5, 5, 5, 2, 3, 4], [0, 7, 6, 2, 5, 7, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 6, 7, 6, 7, 7, 6, 1], [6, 1, 2, 3, 3, 7, 5, 3], [0, 7, 6, 2, 5, 2, 3, 4], [1, 7, 5, 5, 5, 7, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 1, 2, 3, 3, 7, 5, 1], [6, 6, 7, 6, 7, 7, 6, 3], [1, 7, 5, 5, 5, 7, 3, 4], [0, 7, 6, 2, 5, 2, 0, 3]]
------------------------
进行变异，变异后的种族为：
[[6, 1, 2, 3, 3, 7, 5, 0], [6, 6, 7, 6, 7, 7, 6, 7], [1, 7, 5, 5, 5, 7, 0, 4], [0, 1, 6, 2, 5, 2, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 6, 7, 6, 7, 7, 6, 0], [6, 1, 2, 3, 3, 7, 5, 7], [0, 7, 5, 5, 5, 7, 0, 4], [1, 1, 6, 2, 5, 2, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 1, 2, 3, 3, 7, 5, 0], [6, 6, 7, 6, 7, 7, 6, 7], [1, 1, 6, 2, 5, 7, 0, 4], [0, 7, 5, 5, 5, 2, 0, 3]]
------------------------
进行杂交，杂交后的种族为：
[[6, 6, 7, 3, 3, 7, 5, 0], [6, 1, 2, 6, 7, 7, 6, 7], [0, 1, 6, 2, 5, 7, 0, 4], [1, 7, 5, 5, 5, 2, 0, 3]]
------------------------
进行变异，变异后的种族为：
[[6, 6, 7, 3, 3, 7, 5, 0], [6, 1, 2, 6, 4, 7, 6, 7], [3, 1, 6, 2, 5, 7, 0, 4], [1, 7, 5, 5, 5, 2, 0, 3]]
----成功找到最优解----
[3, 1, 6, 2, 5, 7, 0, 4]
```

遗传算法相比于迭代改进法和模拟退火法，优化之处在于总能找到一个八皇后问题的可行解。