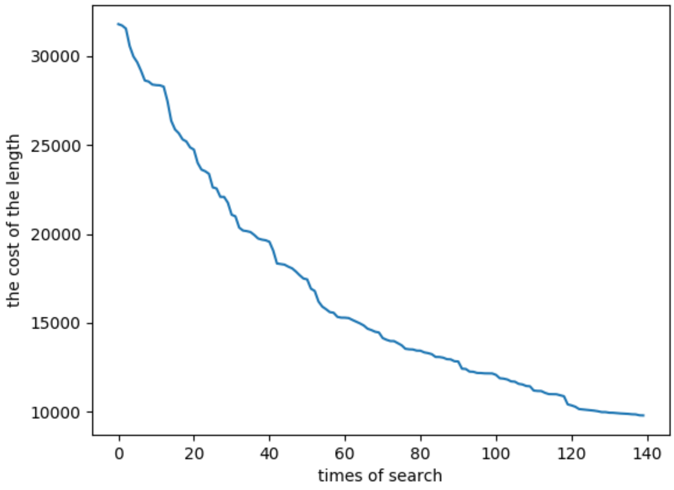
人工智能基础作业 3

张磊 2017K8009922027

（代码位于文末附录处）

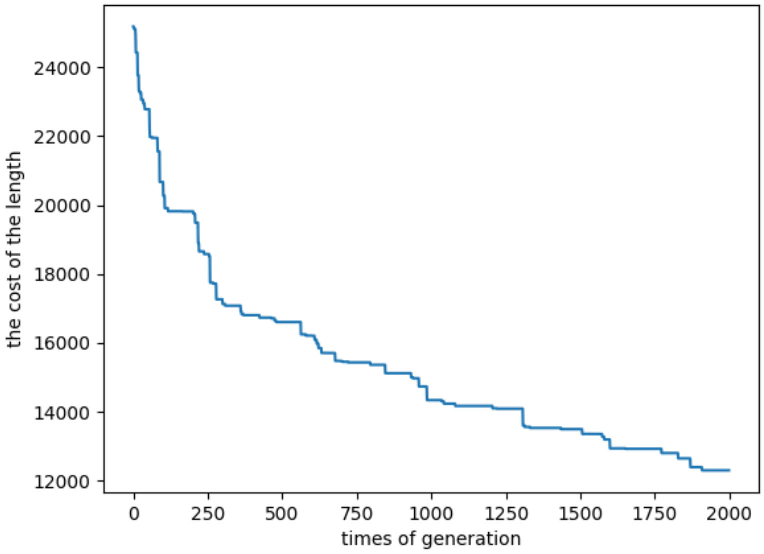
4.3 解：

a. TSP 问题爬山法

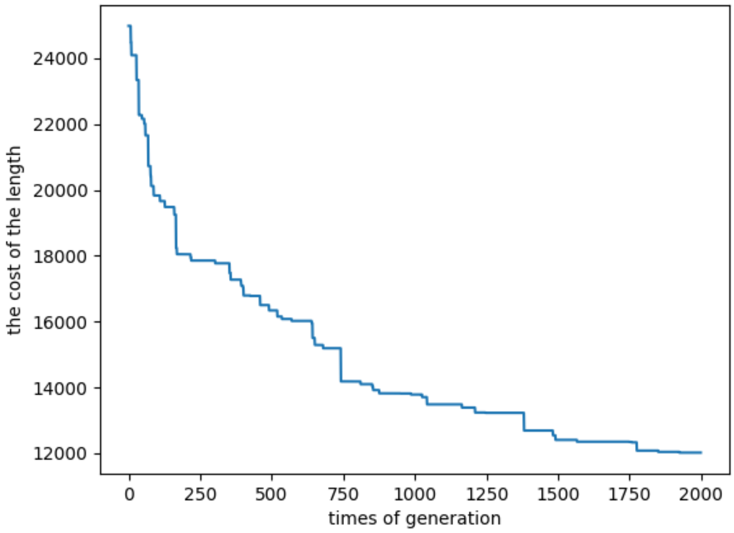


b. TSP 遗传算法：

(种群数量 100，杂交率 0.2，变异率 0.2，繁殖 2000 代)



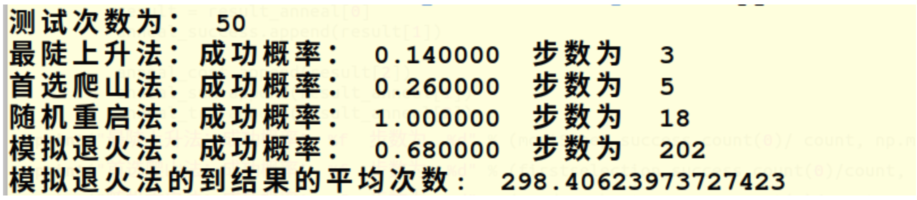
（种群数量 200，杂交率 0.2，变异率 0.2，繁殖 2000 代）



4.4 解：

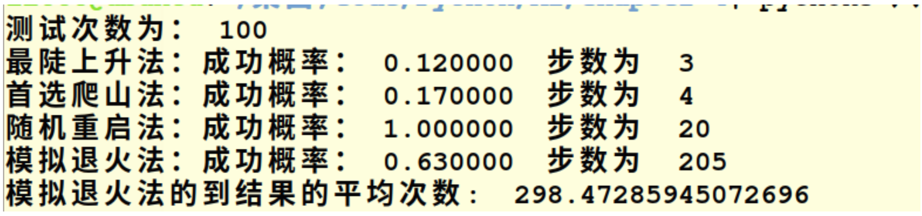
a. 八皇后问题

测试 50 次：



```
测试次数为： 50
最陡上升法： 成功概率：  0.140000  步数为  3
首选爬山法： 成功概率：  0.260000  步数为  5
随机重启法： 成功概率：  1.000000  步数为  18
模拟退火法： 成功概率：  0.680000  步数为  202
模拟退火法的到结果的平均次数：  298.40623973727423
```

测试 100 次：

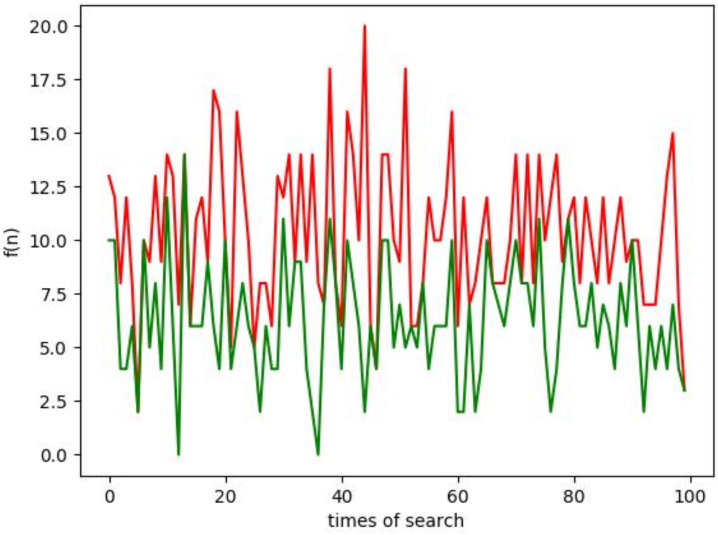


```
测试次数为：  100
最陡上升法： 成功概率：  0.120000  步数为  3
首选爬山法： 成功概率：  0.170000  步数为  4
随机重启法： 成功概率：  1.000000  步数为  20
模拟退火法： 成功概率：  0.630000  步数为  205
模拟退火法的到结果的平均次数：  298.47285945072696
```

可见，最陡上升法成功率最低，随机重启法成功概率最高；

b. 八数码问题

首选爬山法结果：



备注：感谢郭豪同学的帮助！

附录 1：八皇后问题代码（Python）:

```
import numpy as np
import random
import math

L = np.random.randint(0, 8, size = 8)


def evaluate(L):

    h = 0
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[i] == L[j]:
                h = h + 1

            offset = j - i
            if abs(L[i] - L[j]) == offset:
                h = h + 1
    return h

def get_score(L):
    score = {}

    for row in range(len(L)):

        for col in range(len(L)):
            if(col == L[row]):

                continue
            else:
                L_move = L.copy()
                L_move[row] = col
                score[(row, col)] = evaluate(L_move)
    return score


def get_next_best(score, L):
    res = []
    for key, value in score.items():
        if value == min(score.values()):
            row = key[0]
            if(L[row] == key[1]):
                continue
            else:
                min_key = key
                res.append(key)
    return res



def get_best_next(L1):
    L = L1.copy()
    depth = 0
    h = evaluate(L)
    score = get_score(L)
    while(h > min(score.values())):
        depth = depth + 1
        next = get_next_best(score, L)
        if(next == []):
            break
        pos = next[0]
        row = pos[0]
        L[row] = pos[1]
        h = evaluate(L)
        score = get_score(L)
        if(h <= min(score.values())):
            break

    result = [L, h, depth]
    return result




def get_next_better(score, L):
    res = []
    h = evaluate(L)
    for key, value in score.items():
        if (value < h):
            row = key[0]
            if(L[row] == key[1]):
                continue
            else:
                min_key = key
                res.append(key)
    return res
```

```
def get_better_next(L1):
    L = L1.copy()
    depth = 0
    h = evaluate(L)
    score = get_score(L)
    while(h > min(score.values())):
        depth = depth + 1
        next = get_next_better(score,L)
        pos = random.choice(next)
        row = pos[0]
        L[row] = pos[1]
        h = evaluate(L)
        score = get_score(L)
        if(h <= min(score.values())):
            break
    result = [L, h, depth]
    return result


def random_restart(L):
    L1 = L.copy()
    depth = 0
    result = get_best_next(L1)
    depth = depth + result[2]
    while(result[1] != 0):
        L1 = np.random.randint(0, 8, size = 8)
        result = get_best_next(L1)
        depth = depth + result[2]

    result[2] = depth

    return result


def Get_Random_Neighbour(L):
        L1 = L.copy()
        while((L1 == L).all()):
            i = random.randint(0, 63)
            k = i // 8
            L1[k] = i % 8
        return L1

def moni(L, count):
    L1 = L.copy()
    L2 = L1.copy()
    h = evaluate(L1)
    limit = 300
    i = 0
    sum_depth = 0
    result = [L2, h, 0]
    while(i < count):
        if(result[1] > h):
            result = [L2, h, depth]
        if(h == 0):
            break
        depth = 0
        i = i + 1
        while(True):
            if(h == 0):
                break
            elif(depth >= limit):
                break

            L2 = Get_Random_Neighbour(L1)
            delta = h - evaluate(L2)

            if(delta > 0):
                L1 = L2
                depth = depth + 1
                sum_depth = sum_depth + 1
                h = evaluate(L1)

            else:
                probability = math.exp(delta/h)
                if(probability > random.random()):
                    L1 = L2
                    depth = depth + 1
                    sum_depth = sum_depth + 1
                    h = evaluate(L1)
    final_result = [result, sum_depth, i]
    return final_result

def test_main(count):

    print("测试次数为：",count)

    mostSteep_success = []
    mostSteep_cost = []
    firstselection_success = []
    firstselection_cost = []
    randomreboot_success = []
    randomreboot_cost = []
    anneal_try = []
    anneal_sum = []
    anneal_success = []
    anneal_cost = []
    i = 0
    while(i < count):
        i += 1
        L = np.random.randint(0, 8, size = 8)

        result_moststeep = get_best_next(L)

        mostSteep_success.append(result_moststeep[1])
        mostSteep_cost.append(result_moststeep[2])

        result_firstselection = get_better_next(L)
        firstselection_success.append(result_firstselection[1])
        firstselection_cost.append(result_firstselection[2])

        result_random = random_restart(L)
        randomreboot_success.append(result_random[1])
        randomreboot_cost.append(result_random[2])



        result_anneal = moni(L,100)
        result = result_anneal[0]
        anneal_success.append(result[1])

        anneal_cost.append(result[2])
        anneal_sum.append(result_anneal[1])
        anneal_try.append(result_anneal[2])
    print("最陡上升法：成功概率：  %f   步数为   %d" %
(mostSteep_success.count(0)/ count, np.mean(mostSteep_cost)))
    print("首选爬山法：成功概率：  %f   步数为   %d" %
(firstselection_success.count(0)/count, np.mean(firstselection_cost)))
    print("随机重启法：成功概率：  %f   步数为   %d" %
(randomreboot_success.count(0)/count, np.mean(randomreboot_cost)))
    print("模拟退火法：成功概率：  %f   步数为   %d" %
(anneal_success.count(0)/count, np.mean(anneal_cost)))
    print("模拟退火法的到结果的平均次数: ", np.sum(anneal_sum)/
np.sum(anneal_try))
test_main(50)
```

附录 2：八码问题代码（Python）：

```
import random
import sys
sys.setrecursionlimit(10000)
import numpy as np
import matplotlib.pyplot as plt
def Judge_success(status):
    success_status = np.arange(0, 9, 1).reshape(3,3)

    if((status== success_status).all()):
        return 1
    else:
        return 0

def Get_Index(state,number):
    for i in range(0, 3):
        for j in range(0, 3):
            if(state[i][j] == number):
                return [i, j]
            else:
                pass

#h1 计算不匹配的数量
def h1(state):
    success_status = np.arange(0, 9, 1).reshape(3,3)
    result = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                continue
            else:
                number = state[i][j]
                loc = Get_Index(success_status,number)
                #print(loc, i, j,number)
                result =result +    abs(i - loc[0]) + abs(j - loc[1])
    return result

def Get_child(state):
    L = []
    loc = Get_Index(state, 0)
    if(loc == [0,0]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_1[0,0], Child_1[0,1] = state[0,1], state[0,0]
        L.append(Child_1)
        Child_2[0,0], Child_2[1,0] = state[1, 0],state[0,0]
        L.append(Child_2)
        return L
    elif(loc == [0, 2]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_1[0,2],Child_1[0,1] = state[0,1], state[0,2]
        L.append(Child_1)
        Child_2[0,2], Child_2[1,2] = state[1, 2], state[0,2]
        L.append(Child_2)
        return L
    elif(loc == [2,0]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_1[2,0],Child_1[2,1] = state[2,1], state[2,0]
        L.append(Child_1)
        Child_2[2,0], Child_2[1,0] = state[1,0], state[2,0]
        L.append(Child_2)
        return L
    elif(loc == [2,2]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_1[2,2],Child_1[2,1] = state[2,1], state[2,2]
        L.append(Child_1)
        Child_2[2,2], Child_2[1,2] = state[1, 2], state[2,2]
        L.append(Child_2)
        return L
    elif(loc == [0,1]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_3 = state.copy()
        Child_1[0,1],Child_1[0,0] = state[0,0], state[0,1]
        L.append(Child_1)
        Child_2[0,1], Child_2[0,2] = state[0, 2], state[0,1]
        L.append(Child_2)
        Child_3[0,1], Child_3[1,1] = state[1,1], state[0,1]
        L.append(Child_3)
        return L
    elif(loc == [1,0]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_3 = state.copy()
        Child_1[1,0],Child_1[0,0] = state[0,0], state[1,0]
        L.append(Child_1)
        Child_2[1,0], Child_2[2,0] = state[2, 0], state[1,0]
        L.append(Child_2)
        Child_3[1,0], Child_3[1,1] = state[1,1], state[1,0]
        L.append(Child_3)
        return L
    elif(loc == [1,2]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_3 = state.copy()
        Child_1[1,2],Child_1[0,2] = state[0,2], state[1,2]
        L.append(Child_1)
        Child_2[1,2], Child_2[2,2] = state[2, 2], state[1,2]
        L.append(Child_2)
        Child_3[1,2], Child_3[1,1] = state[1,1], state[1,2]
        L.append(Child_3)
        return L
    elif(loc == [2,1]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_3 = state.copy()
        Child_1[2,1],Child_1[2,0] = state[2,0], state[2,1]
        L.append(Child_1)
        Child_2[2,1], Child_2[2,2] = state[2, 2], state[2,1]
        L.append(Child_2)
        Child_3[2,1], Child_3[1,1] = state[1,1], state[2,1]
        L.append(Child_3)
        return L
    elif(loc == [1,1]):
        Child_1 = state.copy()
        Child_2 = state.copy()
        Child_3 = state.copy()
        Child_4 = state.copy()
        Child_1[1,1],Child_1[1,0] = state[1,0], state[1,1]
        L.append(Child_1)
        Child_2[1,1], Child_2[0,1] = state[0, 1], state[1,1]
        L.append(Child_2)
        Child_3[1,1], Child_3[1,2] = state[1,2], state[1,1]
        L.append(Child_3)
        Child_4[1,1], Child_4[2,1] = state[2,1], state[1,1]
        L.append(Child_4)
        return L
    else:
        pass

def Judge_in(child, L_t, L1):
    result = 0
    for item in L_t:
        if((child[0] == item[0]).all()):
            return 1
    for item in L1:
        if((child[0] == item[0]).all()):
            return 1
    return result

def f(n):
    return n[1] + n[2]
    #return n[2]
    #return n[2]
def insert_temp(temp, L,L1):
    if(Judge_in(temp, L,L1)):
        return L
    f_temp = f(temp)
    l = len(L)
    i = 0
    while i < l:
        f_status = f(L[i])
        if(f_temp < f_status): #等于时优先级不同
            L.insert(i, temp)
            return L
        i = i + 1
    L.append(temp)
    return L

def Getbestchild(L):
    if(len(L) == 0):
        return []
    best    = L[0].copy()
    h_best    = f(best)
    index = 0
    for i in range(1, len(L)):
        if(L[i][2] == 0):
            return [L[i], i]
        if(f(L[i]) < h_best):
            best = L[i].copy()
            h_best = f(L[i])
            index = i
    return [best, index]

#误差最小的元素
def Getbestresult(L):
    best = L[0].copy()

    for item in L:
        if item[2] < best[2]:
            best = item.copy()
    return best

#最陡爬山法
def mostSteepClimb(L, L1, max_time):
    cstatus = L[0].copy()
    L1.append(cstatus)
    del L[0]
    hc = f(cstatus)
    count = 0
    e_count = 0
    while(cstatus[2] != 0):
        chlid_list = Get_child(cstatus[0])
        for item in chlid_list:
            child = [item, cstatus[1] + 1, h1(item)]
            if(~Judge_in(child, L, L1)):
                L.append(child)
        best = Getbestchild(L)
        '''
        if(best[0][2] == cstatus[2]):
            e_count = e_count + 1
            if(e_count > 50):
                return [cstatus, L1]
        '''
        nstatus = best[0]
        index = best[1]
        if(nstatus[2] == 0):
            L1.append(nstatus.copy())
            del L[index]
            return [nstatus,L1]
        if(count <= max_time):
            count = count + 1
            L1.append(nstatus.copy())
            del L[index]
            cstatus = nstatus.copy()
        else:
            #print(L1)
            result = Getbestresult(L1)
            return [result, L1]
    return [cstatus, L1]

def get_better_next(L, nstatus):
    result = []
    for i in range(len(L)):
        if f(L[i]) < f(nstatus) + 2:
        #if L[i][2] < nstatus[2] + 1:
            #print("one better choice")
            result.append(i)
    return result


#首选爬山法

def first_selection(L, L1, max_time):
    cstatus = L[0]
    L1.append(cstatus.copy())
    del L[0]
    count = 0
    while(cstatus[2] != 0):
        chlid_list = Get_child(cstatus[0])
        for item in chlid_list:
            child = [item, cstatus[1] + 1, h1(item)]
            if(~Judge_in(child, L, L1)):
                L.append(child)
        r = get_better_next(L, cstatus)
        if(len(r) == 0):
            #print("no better choice", count)
            best = Getbestresult(L1)
            return [best,L1]
        index = random.choice(r)
        cstatus = L[index]
        #print(count)
        count = count + 1
        L1.append(cstatus.copy())
        del L[index]
        if(count > max_time):
            best = Getbestresult(L1)
            return [best,L1]
    return [cstatus, L1]

def main():
    test = []
    i = 0

    while(i < 100):
        init_status = np.arange(0, 9, 1)
        np.random.shuffle(init_status)
        init_status = init_status.reshape(3,3)
        test.append(init_status)
        i = i + 1

    result_moststeep = []
    result_firstselection = []
    moststeep_cost = []
    firstselection_cost = []
    for item in test:
        init_status = item.copy()
        L = []
        L.append([init_status, 0 , h1(init_status)])
        L1 = []
        s = mostSteepClimb(L, L1, 300)
        moststeep_cost.append(len(s[1]))
        result_moststeep.append(s[0][2])
        init_status = item.copy()
        L = []
        L.append([init_status, 0 , h1(init_status)])
        L1 = []
        s = first_selection(L, L1, 300)
        firstselection_cost.append(len(s[1]))
        result_firstselection.append(s[0][2])

    plt.plot(np.array(result_firstselection),'r' )
    plt.plot(np.array(result_moststeep), 'g')
    plt.ylabel("f(n)")
    plt.xlabel("times of search")
    plt.show()

main()
```

附录 3：旅行商问题（遗传算法）：

```
import numpy as np
import matplotlib.pyplot as plt
import random

#从 CSDN 上找到的数据

coordinates =
np.array([[565.0,575.0],[25.0,185.0],[345.0,750.0],[945.0,685.0],[845.0,655.0],

[880.0,660.0],[25.0,230.0],[525.0,1000.0],[580.0,1175.0],[650.0,1130.0],
                              [1605.0,620.0],[1220.0,580.0],[1465.0,200.0],[1530.0,
5.0],[845.0,680.0],

[725.0,370.0],[145.0,665.0],[415.0,635.0],[510.0,875.0],[560.0,365.0],

[300.0,465.0],[520.0,585.0],[480.0,415.0],[835.0,625.0],[975.0,580.0],

[1215.0,245.0],[1320.0,315.0],[1250.0,400.0],[660.0,180.0],[410.0,250.0],

[420.0,555.0],[575.0,665.0],[1150.0,1160.0],[700.0,580.0],[685.0,595.0],

[685.0,610.0],[770.0,610.0],[795.0,645.0],[720.0,635.0],[760.0,650.0],

[475.0,960.0],[95.0,260.0],[875.0,920.0],[700.0,500.0],[555.0,815.0],
                              [830.0,485.0],[1170.0,
65.0],[830.0,610.0],[605.0,625.0],[595.0,360.0],
                              [1340.0,725.0],[1740.0,245.0]])


#得到距离矩阵的函数

def getdistmat(coordinates):

     num = coordinates.shape[0] #52 个坐标点

     distmat = np.zeros((52,52)) #52X52 距离矩阵

     for i in range(num):
          for j in range(i,num):
            distmat[i][j] = distmat[j][i]=np.linalg.norm(coordinates[i]-coordinates[j])
    return distmat


distmat = getdistmat(coordinates)

def evaluate(L):
    length = 0
    for i in range(len(L) - 1):
        length += distmat[L[i]][L[i+1]]
    return length



#初始化一个 group
def init_group(size):
    L = np.arange(0, 52)
    initial_group = []
    #生成初始群体
    for i in range(size):
        L1 = L.copy()
        np.random.shuffle(L1)
        initial_group.append(L1)
    return init_group

#选择
def Selection(group):
    group_copy = []
    for L in range(len(group)):
        h = evaluate(L)
        if(h < 12500):
            group_copy.append(L)
    return group_copy

#交叉
def Cross(parent1, parent2):
    index1 = random.randint(0, len(parent1) - 1)
    #print(index1)
    index2 = random.randint(index1, len(parent2) - 1)
    #print(index2)
    temp = parent2[index1 : index2]
    child = []
    i = 0
    for k in parent1:
        if i == index1:
            child.extend(temp)
            i += 1
        if k not in temp:
            child.append(k)
            i += 1
    return child

#编译
def Mutation(L):
    L1 = L.copy()
    index1 = random.randint(0, len(L) - 1)
    index2 = random.randint(0, len(L) - 1)
    if (index1 != index2):
        L1[index1], L1[index2] = L1[index2], L1[index1]
    return L1

def get_newchild(group, crossrate, mutationrate):
    rate = random.random()
    i = random.randint(0, len(group) - 1)
    parent1 = group[i]
    #按照概率交叉
    if(rate < crossrate):
        i = random.randint(0, len(group) - 1)
        parent2 = group[i]
        child = Cross(parent1, parent2)
    else:
        child = parent1
    #按照概率编译
    rate = random.random()
    if (rate < mutationrate):
        child = Mutation(child)

    return child

#获取最好的元素
def get_best(group):
    best = group[0].copy()
    h_best = evaluate(best)
    for L in group:
        h = evaluate(L)
        if (h < h_best):
            best = L.copy()
            h_best = h
    return best

#产生下一代
def next_generation(group, crossrate, mutationrate):
    new_group = []
    best = get_best(group)
    new_group.append(best)
    #print(evaluate(best))
    while (len(new_group) < 100):
        child = get_newchild(group, crossrate, mutationrate)
        new_group.append(child)
    return new_group

def test_main(times, crossrate, mutationrate):
    L = np.arange(0, 52)
    initial_group = []
    #生成初始群体100个
    for i in range(200):
        L1 = L.copy()
        np.random.shuffle(L1)
        initial_group.append(L1)
    i = 0
    result = []
    while(i < times):
        i = i + 1
        initial_group = next_generation(initial_group, crossrate, mutationrate)
        best = get_best(initial_group)
        result.append(best)

    leng = []
    for i in range(len(result)):
        h = evaluate(result[i])
        leng.append(h)
    best = get_best(result)
    #print("the final method is ", best)
    plt.plot(np.array(leng))
    plt.ylabel("the cost of the length")
    plt.xlabel("times of generation")
    plt.show()

test_main(2000,0.2,0.2)
```

附录 4：旅行商问题（爬山法算法）:

```
import numpy as np
import matplotlib.pyplot as plt
import random

#从 CSDN 上找到的数据

coordinates =
np.array([[565.0,575.0],[25.0,185.0],[345.0,750.0],[945.0,685.0],[845.0,655.0],

[880.0,660.0],[25.0,230.0],[525.0,1000.0],[580.0,1175.0],[650.0,1130.0],
                              [1605.0,620.0],[1220.0,580.0],[1465.0,200.0],[1530.0,
5.0],[845.0,680.0],

[725.0,370.0],[145.0,665.0],[415.0,635.0],[510.0,875.0],[560.0,365.0],

[300.0,465.0],[520.0,585.0],[480.0,415.0],[835.0,625.0],[975.0,580.0],

[1215.0,245.0],[1320.0,315.0],[1250.0,400.0],[660.0,180.0],[410.0,250.0],

[420.0,555.0],[575.0,665.0],[1150.0,1160.0],[700.0,580.0],[685.0,595.0],

[685.0,610.0],[770.0,610.0],[795.0,645.0],[720.0,635.0],[760.0,650.0],

[475.0,960.0],[95.0,260.0],[875.0,920.0],[700.0,500.0],[555.0,815.0],
                              [830.0,485.0],[1170.0,
65.0],[830.0,610.0],[605.0,625.0],[595.0,360.0],
                              [1340.0,725.0],[1740.0,245.0]])


#得到距离矩阵的函数

def getdistmat(coordinates):

    num = coordinates.shape[0] #52 个坐标点

    distmat = np.zeros((52,52)) #52X52 距离矩阵

    for i in range(num):
        for j in range(i,num):
            distmat[i][j] = distmat[j][i]=np.linalg.norm(coordinates[i]-coordinates[j])
    return distmat


distmat = getdistmat(coordinates)

def evaluate(L):
    length = 0
    for i in range(len(L) - 1):
        length += distmat[L[i]][L[i+1]]
    i = len(L)
    length += distmat[L[i-1]][L[0]]
    return length

'''

1. 随机换两个点

2. 随机将 2 个点之间的结点逆序

2. 随机选择 3 个结点，m, n, k, 将结点吗， 你间结点移动到 k 后。

'''

#首选爬山法

def get_neighbor(L):
    h = evaluate(L)
    result= []
    for i in range(len(L)):
        for j in range(i,len(L), 1):
            if(i == j):
                continue
            L_copy = L.copy()
            temp = L_copy[i]
            L_copy[i] =L_copy[j]
            L_copy[j] = temp
            h_copy = evaluate(L_copy)
            if(h_copy < h):
                result.append([i, j])
    return result

def firstselect(L):
    h = evaluate(L)
    L_copy = L.copy()
    result = []
    while(1):
        res = get_neighbor(L_copy)
        if(len(res) == 0):
            print(L_copy)
            break
        pos = random.choice(res)
        i = pos[0]
        j = pos[1]
        temp = L_copy[i]
        L_copy[i] =L_copy[j]
        L_copy[j] = temp
        L1 = L_copy.copy()
        result.append(L1)
    return result



def test_main():
    L = np.arange(0, 52)
    np.random.shuffle(L)
    print("the initial solution is ", L)
    result = firstselect(L)
    leng = []

    for i in range(len(result)):
        h = evaluate(result[i])
        leng.append(h)
    print("the final method is ", result[len(result) - 1])
    plt.plot(np.array(leng))
    plt.ylabel("the cost of the length")
    plt.xlabel("times of search")
    plt.show()

test_main()
```