武汉大学计算机学院

本科生课程设计报告

人工智能引论大作业之 TSP 的解决算法

专 业 名 称 ：CS

课 程 名 称 ：人工智能引论

指 导 教 师 一：AY

学 生 学 号 ：

学 生 姓 名 ：JaeHua

二〇二三年十一月

郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：JaeHua 日期：2023 年 11 月 4 日星期六

摘要

TSP算法解决实验的实验目的是用经典搜索算法（A*算法或其他算法）和智能算法（遗传算法，免疫算法或其他算法）两种算法来解决。。

实验设计主要遵循控制变量法，以对比不同算法的效率、精度等指标。

实验内容主要包括:

1. 使用贪心算法,根据距离最近的未访城市问来构造路径。

2. 使用蚁群算法,设计信息素启发式函数,实现状态转移和信息素更新。

实验结论为:

1. 贪心算法可以快速找到可行解,但容易陷入局部最优。

2. 蚁群算法可以逐步接近最优解,但收敛速度相对缓慢。结论:

贪心算法适合快速找出可行解;蚁群算法适合持续逼近最优解。两算法优势互补。

……

……

关键词：TSP；贪心算法；蚁群算法

目录

1 实验背景与目的

1.1 实验背景

TSP 全称是 Traveling Salesman Problem,中文翻译为旅行商问题或行商问题。它是一个著名的 NP 困难的组合优化问题。问题描述如下:给出多个城市和每对城市之间的距离,旅行商需要从一个城市出发,途经其他所有城市一次且仅一次,然后回到起始城市。旅行的目标是找到一条路径,使得旅行的总距离最短。这个问题在实际中有许多应用,如:送货或邮递路线规划、机器人或数控机床的运动规划、 DNA 测序仪的工作路径优化、电路板钻孔顺序的规划、车间流水线的布局设计等等。

解决 TSP 问题的算法有:精确算法:动态规划、分支定界法等。近似算法:贪心最近邻算法、蚁群算法、遗传算法等。启发式算法:2-opt、3-opt。TSP 问题本身属于 NP 难问题,无法在多项式时间内获得精确解。对大规模 TSP 问题,一般使用近似算法和启发式算法来求解。这个问题很好的考察了算法设计技巧和计算机求解组合优化问题的能力。它有着广泛的应用前景和研究价值。

本实验利用贪心算法和蚁群算法两种不同思路的算法来解决 TSP 问题, 两种算法从不同的角度和思路出发,但目的都在于解决 TSP 问题。本实验将通过比较,找出两种算法各自的优势及不足,为算法设计提供借鉴

1.2 算法介绍

1.2.1 贪心算法

贪心算法是一种常用于求解优化问题的启发式算法。其基本思想是:在每一步的决策中,都选择局部最优解,从而希望能达到全局最优。贪心算法具有以下几个主要特点:每一步都做出一个局部最优的选择,不回退; 整体的解不一定是全局最优解; 算法比较简单、快速。

其适用于找出近似最优解的场景对于 TSP 问题,贪心算法是一个简单有效的近似算法。相比精确算法,贪心算法的优点是速度快,实现简单,对大规模问题仍然能求出可接受的解,适合用于在线求解或对时间效率有要求的场景。其劣势是无法保证找到最优解,解的质量可能略低于精确算法。

1.2.2 蚁群算法

蚁群算法(Ant Colony Optimization, ACO)是一种基于蚂蚁寻找食物的行为而设计的随机优化算法。其基本思想是:1. 蚂蚁在搜索食物时,会在路径上留下信息素;2. 后来的蚂蚁会根据信息素量选择路径,选择概率与信息素量成正比;3. 按照正反馈机制,信息素越多的路径被选择的概率越大,被选择的次数越多,信息素也越多;4. 通过迭代,信息素会积累在最短路径上,蚂蚁最终找到该路径。相比贪心算法,蚁群算法利用概率搜索和信息交互得到更好的解。它可以有效解决 TSP 问题,避免陷入局部最优,适合求解复杂的组合优化问题。

1.3 实验目的

1.3.1 使用经典搜索贪心算法解决 TSP 问题

本实验目的一是基于贪心算法的思想,设计并实现一种快速找到 TSP 问题的可行解的算法。贪心算法的核心思想是每一步都做出一个局部最优选择,期望通过局部最优能达到全局最优。本实验将探索如何设计贪心选择策略,并评估这种算法的时间复杂度及解的质量。

1.3.2 目的二:使用智能搜索蚁群算法解决 TSP 问题

本实验目的二是基于蚁群算法的思想,设计并实现一种可以稳定逼近 TSP 问题最优解的算法。蚁群算法通过模拟蚂蚁的路径选择方式,不断更新信息素,让蚂蚁逐步朝着最短路径集中。本实验将研究信息素设计与更新策略,并评估算法在求解最优解方面的性能。

2 实验设计

2.1 贪心算法设计

1. 贪心选择策略设计:选择距离最近的城市作为下一个访问城市,因为距离最近可以快速构建路径,效率高,容易找到近似解。同时,距离最近也符合人的经验,人总倾向于选择离当前位置较近的下一个目的地。

2. 路径构造实现:从任意给定城市开始,初始化路径为只包含起点城市。然后遍历所有未访问城市,计算它们与当前最后一个城市的距离,选择距离最近的未访问城市作为下一个城市,将其添加到路径末尾。重复此过程,直到路径包含了所有城市为止。在迭代过程中,需要实时更新当前路径长度,并与已得到的最优解进行比较,记录较优解。整个构造路径过程效率较高,可快速得到一个可行遍历解。但贪心选择仅根据局部距离,可能会导致全局路径不优。

3. 重复运行:从不同随机起点开始构造路径,重复多次运行,记录最优解。

4. 绘图，绘制出路线图以及随迭代次数最优解的变化，便于更直观的分析。

2.2 蚁群算法设计

1. 信息素表示设计:在每对城市间路径上都设置一个信息素变量 τij 表示该路径的信息素浓度,初始值均相同。信息素浓度越高表示路径越可取。
2. 状态转移规则:蚂蚁从当前城市移动到下一城市时,根据所有可选城市路径的信息素浓度概率分布,通过轮盘赌方法选择下一城市。概率计算公式为 Pij = τij^α / ∑τij^α。α 为信息素重要程度参数。
3. 信息素更新策略:采用蚂蚁圈系统。一次迭代结束后,在最短路径上的城市间信息素浓度增强 Δτij,其余路径信息素按一定比例挥发。充分利用蚂蚁互相协同的正反馈机制。
4. 算法框架:初始化信息素矩阵 τij,置蚂蚁于起点城市。重复迭代蚂蚁构建解、信息素更新等步骤,直到达到設定迭代次数或路径足够优化。
5. 参数调整:主要调整信息素相关参数,如信息素挥发率、额外增强量Δτ等,控制信息素积累速度,避免搜索过快陷入局部最优。如果初始信息素太小，算法容易早熟，蚂蚁会很快的全部集中到一条局部最优的路径上。反之，如果初始信息素太大，信息素对搜索方向的指导作用太低，也会影响算法性能。

3 实验内容

3.1 贪心算法实验内容

1. 算法实现
①导入所需库:math、random、matplotlib 等
②读取 TSP 数据集,解析出城市坐标到城市列表 citys 计算城市间距离矩阵 dist_map,使用双重循环计算城市间欧式距离- 设置算法参数:迭代次数 num_iter 等
③路径初始化:随机选择起点城市,初始化路径 path 为起点城市
④贪心选择策略:
- 在未访问城市列表 notVis 中,找到距离当前城市 cur_city 最近的城市 next_city
- 将 next_city 插入 path 中的,
- 更新 cur_city 为 next_city-

⑤最优解更新:计算当前路径长度,与历史最优解比较,保存较优解- 迭代终止条件:迭代 num_iter 次后退出

2. 结果输出
打印最终得到的最优路径及长度绘制路径图:从最优路径中提取城市坐标,绘制路径线绘制迭代过程图:画出每次迭代的路径长度变化曲线

3. 时间统计- 记录算法开始和结束时间,计算执行时间

4. 主程序框架:
①导入库,读取数据。②参数设置,辅助函数定义。③迭代循环主体。④路径初始。
⑤贪心构造路径。⑥更新最优解。⑦结果输出和绘图。⑧统计时间

3.2 蚁群算法实验内容

1. 算法实现
①导入库:math、random、matplotlib 等
②读取城市坐标数据,存储在 citys 中
③设置算法参数:信息素挥发率 rho、启发因子 α、β 等。
④计算城市距离矩阵 dist_map
⑤初始化信息素矩阵 pheromone、Visibility 矩阵 vispos

2.迭代搜索框架:
- 路径初始化:随机产生蚂蚁的起点城市
- 构造路径:根据转移概率,轮盘选择下个城市
- 计算路径距离 length
- 更新信息素:在最佳路径增强信息素,其他路径信息素挥发- 关键函数:
- 计算城市间距离 cacl_dist_map
- 轮盘选择 next_city
- 更新信息素 update_pheromone2.

2. 结果输出打印最优路径及长度
3. - 保存结果到文件
4. - 绘制路径图和迭代过程图

5. 时间统计- 记录算法执行开始和结束时间
6. - 计算时间差即为执行时间

5. 主程序框架- 导入库,读取数据
- 参数设置,辅助函数定义
- 迭代框架主体
- 路径初始化
- 构造路径
- 更新信息素
- 结果输出,绘图

4 实验代码分析

4.1 贪心算法代码分析

1. 读取数据

这段代码打开 st70.tsp 文件,读取 NODE_COORD_SECTION 部分的城市坐标数据,分割后存储到 citys 列表中。

```
'''读取数据,记录城市坐标'''
with open('st70.tsp') as f:
    while True:
        line = f.readline()

        if line.startswith("NODE_COORD_SECTION"):
            break
    # 记得加上换行符号
    while line != "EOF\n":
        line = f.readline()
        if line == "EOF\n":
            break
        # 分割成多个字符串
        line = line.split(' ')
        # print(line)
        nums = []
        for num in line[1:]:
            nums.append(int(num))
        citys.append(nums)
```

2. 计算城市距离矩阵

```
'''城市距离矩阵'''
def cacl_dist_map(citys):
    for i in range(len(citys)):
        for j in range(i,len(citys)):
            if(i!=j):
             dist_map[i][j] = dist_map[j][i] = cacl_dist(i,j)
            else:

                dist_map[i][j] = 0
```

这段代码定义了 cacl_dist_map()函数,利用双层循环计算城市距离矩阵 dist_map。

3. 迭代框架

```
python

while cnt < num_iter:

    # 路径生成

    # 计算路径长度

    # 比较更新最优解

    cnt += 1
```

这是迭代搜索的主体框架,通过 num_iter 次迭代逐步逼近最优解。

4. 更新最优解

```
if cnt >=1 and dist_best[cnt-1] < dist_best[cnt]:
    dist_best[cnt] = dist_best[cnt-1]
    path_best[cnt] = path_best[cnt-1]
```

比较当前解与历史最优解,如果更优则更新最优距离 dist_best 和最优路径 path_best。

5. 绘图

调用 matlab 库进行绘制。绘图分为两部分，一个是路径的绘制，还有一个就是最优解随着迭代次数的变化。

4.2 蚁群算法代码分析

1. 参数设置

```
#城市数量
city_count = len(citys)
# 信息素初始值
init_pheromone = 0
# 信息素挥发率
rho = 0.2
# 信息素重要程度参数
alpha = 1
# 期望值启发因子
beta = 5
# 蚁群数量
num_ants = 50
# 迭代次数
num_iter = 1000
#城市距离矩阵
dist_map = [[0]*(len(citys)) for i in range(len(citys))]
#Q为常数
Q = 10
```

2. 能见度矩阵

能见度矩阵用来后续计算蚂蚁选择城市的概率。

```
'''计算可见度矩阵'''
for i in range(len(citys)):
    for j in range(i+1, len(citys)):
        vispos[i][j] = vispos[j][i] = 1 / dist_map[i][j]
```

3. 蚂蚁数量判断

```
python

if num_ants <= city_count:

    # 蚂蚁数<=城市数 设置起点
```

```
else:
    # 蚂蚁数>城市数 随机设置起点
```

每只蚂蚁开始的城市选择都是随机的.

```
cities = list(range(city_count))
#打乱列表
random.shuffle(cities)
# print(cities)
# 设置起点城市
for i in range(num_ants):
    candidate[i][0] = cities[i]
```

根据蚂蚁数量与城市数量的关系,设置蚂蚁起点城市,充分利用蚂蚁资源。

4. 路径构造

```
python
for i in range(1,city_count):
    # 计算转移概率
    # 轮盘选择下一座城市
    # 更新路径距离
```

关于轮盘赌:

```
#轮盘赌
total = 0
for p in posibility:
    total += p
posi = []
summ = 0
#进行切分
for p in posibility:
    summ += p
    posi.append(summ / total)
# if i==0 and j ==1:print(posi)
#生成一个0~1的随机数，用于轮盘赌
num = random.random()
for it in range(len(posi)):
  posi[it] -= num
```

核心路径构造代码,根据转移概率选择下一城市,迭代访问所有城市。

5. 更新信息素

```
'''信息增量矩阵'''
add_pheromone = [[1] * len(citys) for i in range(len(citys))]

for ii in range(num_ants):

    for j in range(city_count - 1):
        add_pheromone[candidate[ii][j]][candidate[ii][j+1]] += Q/length[ii]
    add_pheromone[candidate[ii][city_count-1]][candidate[ii][0]] += Q/length[ii]
```

双重循环,根据蚂蚁路径距离更新相应路径上的信息素。

6. 信息素更新公式

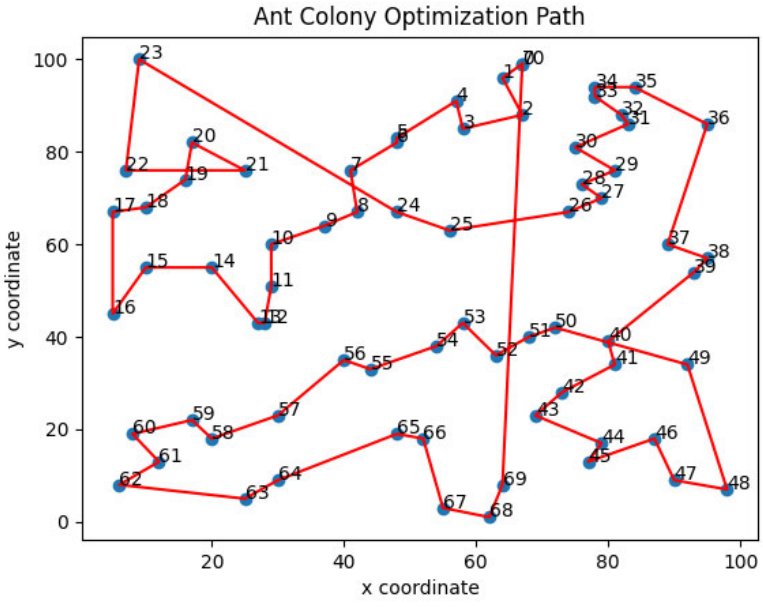
```
'''信息素更新'''
for i in range(len(citys)):
    for j in range(len(citys)):
        #更新公式
        pheromone[i][j] = (1 - rho) * pheromone[i][j] + add_pheromone[i][j]
```

信息素原有部分挥发,新增部分融合,实现信息素的迭代更新。

5 结果分析

5.1 贪心算法结果

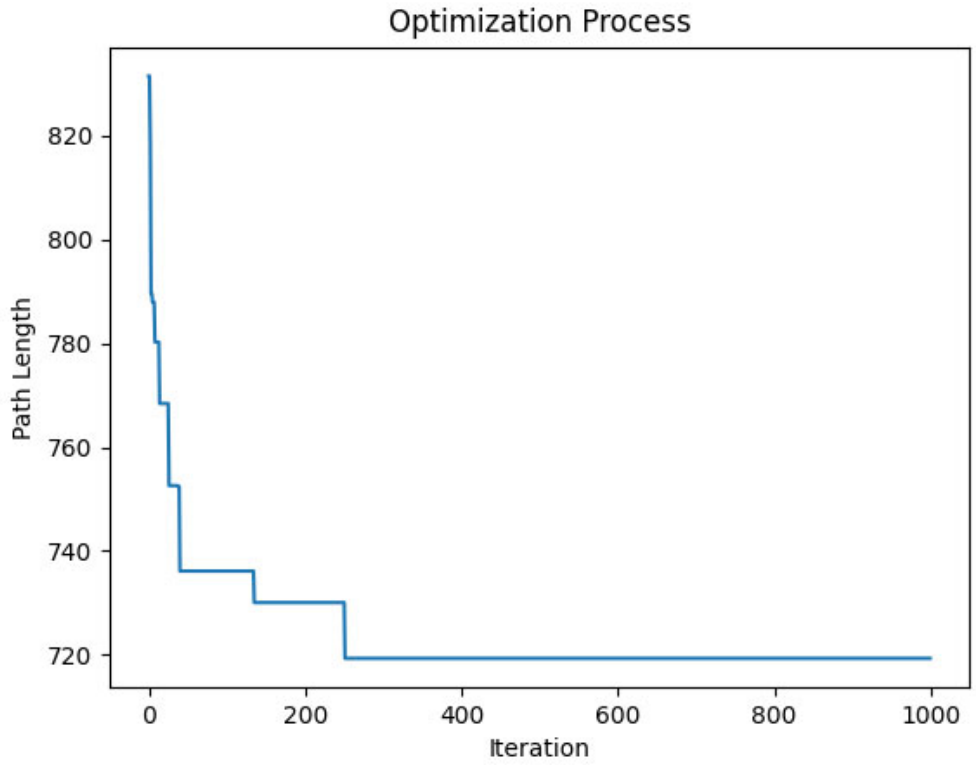
（如图）



图表 1 贪心算法路线图

```
贪心算法的最优路径 [35, 0, 22, 46, 15, 36, 57, 49, 9, 51, 59, 11, 20, 33, 32, 53, 61, 47, 66, 55, 64, 50, 10, 63, 4, 52, 65, 62,
  21, 58, 37, 68, 30, 12, 28, 69, 34, 56, 14, 23, 1, 6, 31, 2, 7, 27, 25, 48, 54, 18, 3, 17, 41, 5, 40, 42, 16, 8, 39, 60,
 38, 44, 24, 45, 26, 67, 43, 29, 19, 13]
迭代 1000 次后 贪心算法求得最优解 719.3178452762141
0.19019031524658203
```

图表 2 贪心算法程序运行结果



图表 3 贪心算法随迭代次数最短路径变化

1. 贪心算法采用每次选择距离最近的未访问城市作为路径选择策略,实现了快速构造解。
2. 由于仅考虑当前信息,而没有整体最优考量,容易陷入局部最优。
3. 结果路径距离与最优解存在一定差距,可以看出贪心算法无法保证找到全局最优解。
4. 但贪心算法时间复杂度较低,适合需要快速求解较优解的场景。
5. 通过设置不同的起点城市,贪心算法可以得到不同的解,然后选择最优的一个。6. 可作为蚁群等元启发式算法的初始化解或者集成进优化框架。
7. 针对规模较小的TSP问题,贪心算法可以取得不错的近似最优解。

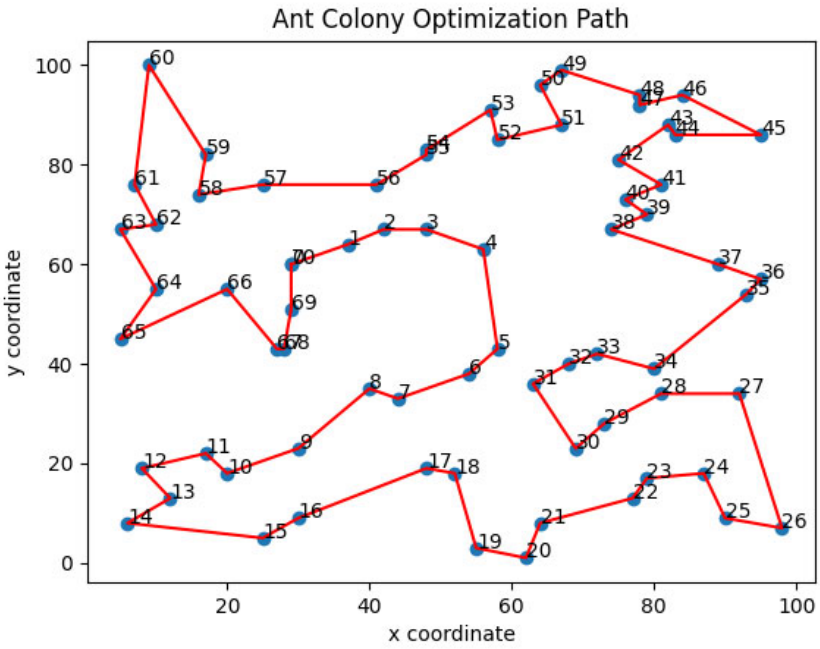
总的来说，算法效率比较高。进行1000次的迭代只用了不到1s，结果也比较近似。

5.2 蚁群算法结果

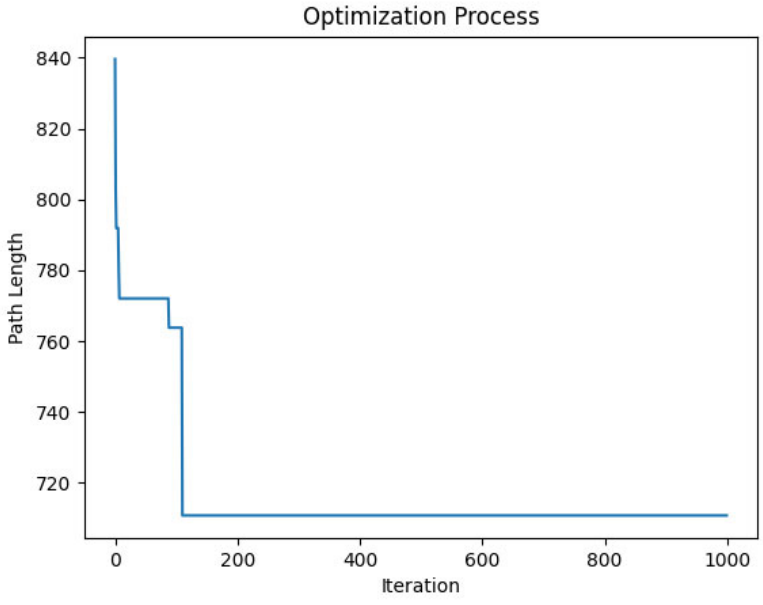
(如图)<目前最好的结果>

```
蚁群算法的最优路径 [59, 51, 9, 4, 52, 5, 40, 42, 16, 8, 39, 60, 38, 44, 24, 45, 26, 67, 43, 29, 19, 13, 27, 7, 25, 48, 54, 18,
 6, 31, 2, 41, 17, 3, 1, 23, 14, 56, 65, 62, 21, 58, 37, 30, 68, 34, 69, 12, 28, 35, 0, 22, 46, 15, 36, 57, 49, 50, 55, 64,
 63, 10, 66, 47, 53, 61, 32, 33, 20, 11]
迭代 1000 次后 蚁群算法求得最优解 685.4501627158198
78.40090870857239
```

图表 4 蚁群算法程序运行结果



图表 5 蚁群算法路线图



图表 6 蚁群算法随迭代次数最短路径变化

在蚁群算法的结果分析中，有两个主要方面需要考虑：精确性和计算时间。

精确性：蚁群算法通常表现出较高的精确性，能够接近全局最优解。这是由于蚂蚁在搜索过程中通过正向和反向的信息素信息交流，有效地引导其他蚂蚁朝着更优解的方向前进。蚂蚁的随机性选择路径也有助于避免陷入局部最优解。

计算时间：蚁群算法的计算时间相对较长，这是由于每只蚂蚁需要在解空间中进行多次迭代

和路径选择。此外，蚁群算法通常需要多次迭代才能收敛到较好的解。因此，与其他优化算法相比，蚁群算法需要更多的计算时间。

6 关于算法的优化

6.1 贪心算法的优化

关于贪心算法的优化我的想法是:在每次从未访问的城市中挑选一个最近的城市的基础上。进一步考虑其在path[]的插入位置的最佳位置。使得path的路径和最小。具体代码可以这么实现:

```
# dis_path += dist_map[cur_city][index]
# if len(path) >= 2:
#     min_dist = float('inf')
#     best_insert_pos = None
#     for i in range(len(path)):
#         temp_path = path[:i] + [index] + path[i:]
#         temp_dist = len(temp_path)
#         if temp_dist < min_dist:
#             min_dist = temp_dist
#             best_insert_pos = i
#     path.insert(best_insert_pos, index)
#     dis_path = len_path(path)
# else:
#     path.append(index)
#     dis_path = len_path(path)
```

经过实测，优化后得到的解与没有优化差别不大，可能是迭代次数多，已经收敛到很好的地步了。

6.2 蚁群算法的优化

6.2.1 调参

通过改变基础参数rho的值，可发现收敛速度随着rho的增加而变快，但是当rho超过0.5后收敛速度反而随着rho的增加而变慢了。并且除了rho=0.5时，其他都出现了函数收敛在局部最优解的情况。分析可知在rho在0.5左右时收敛速度最快并且不容易影响全局最优性，rho过高时会导致出现局部最优解和收敛速度变慢的情况。

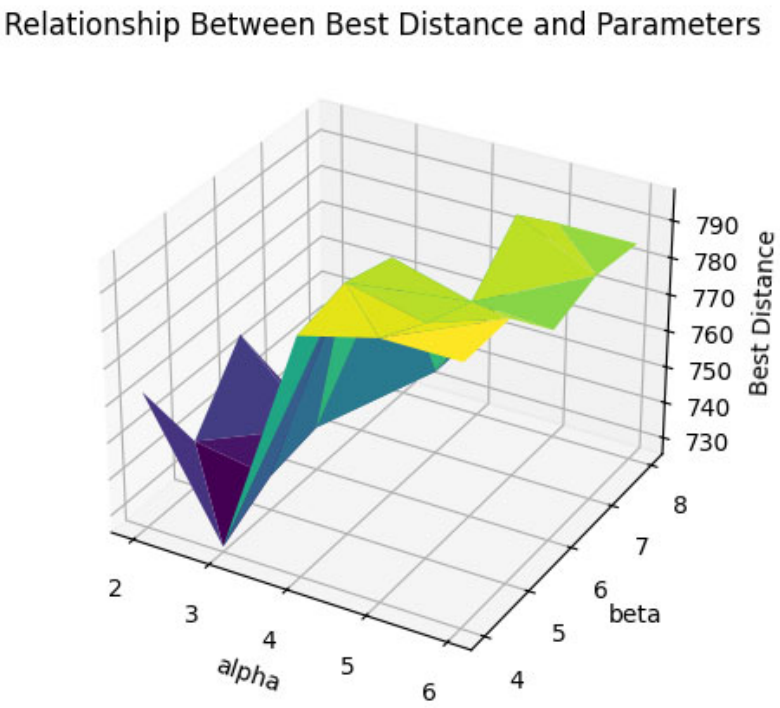
通过改变初始参数alpha的值，发现随着alpha的增大收敛速度逐渐变快，但是从alpha=3后开始到alpha=6这段区间可发现，随着alpha的增大收敛速度的变化却不够明显，虽然仍然在变快，但是却即为缓慢，而且随着alpha的增高函数也过早收敛陷入了局部最优解

通过改变beta的值发现在beta值为3到6之间效果最好，此时随着beta数值的增加收敛速度加快，且不容易陷入局部最优解，而小于3时随着值的减少，找到的解都是局部最优解，

且离全局最优解的距离越来越大，而当beta值大于6时，随着值的增大，收敛速度虽然加快，但是却容易过早收敛陷入局部最优解。
（如图简单展示）

```
# 信息素重要程度参数
alphas = [2, 3, 4, 5,6]
betas = [4,5,6,7,8]
```

图表 7 调参例图



图表 8Alpha 与 Beta 同最优路径立体图

6.2.2 初始化改进

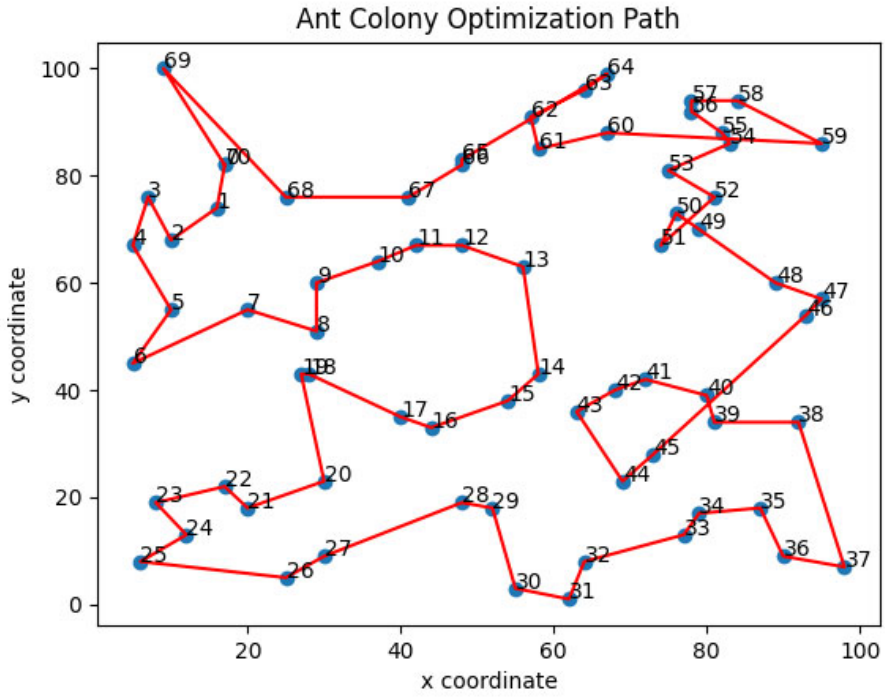
可以结合贪心算法，在初始的选择路径增大信息素的含量。有利于收敛到比较优秀的值。

6.2.3 随机化 alpha,beta

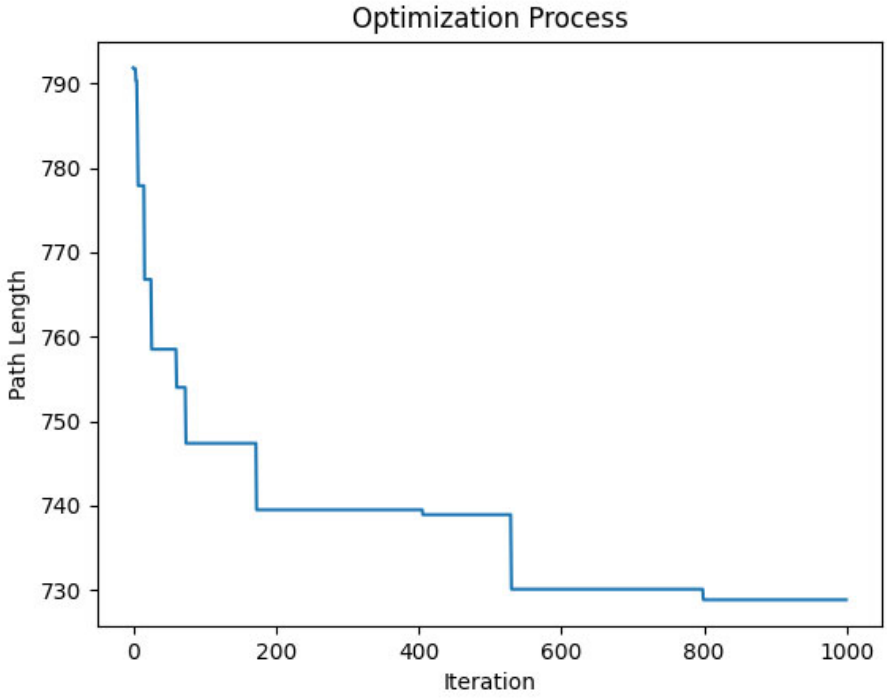
这样可以让解覆盖到更大的区域，为每一只蚂蚁分配不同的alpha,beta，也增大了随机性，有利于找到全局最优解.

```
# beta = [5 * pow(random.random(), 0.6) for _ in range(num_ants)]
# alpha = [3 - random.random() * min(3, 2 + beta[i] / 3) for i in range(num_ants)]
```

图表 9 随机化参数代码



图表 10 随机化后路线



图表 11 随机化后迭代次数与最优路径图

经过实测（结果上图），随机化beta与alpha效果不错，迭代一百次已经达到了不错的效果，有利于找到全局最优解。

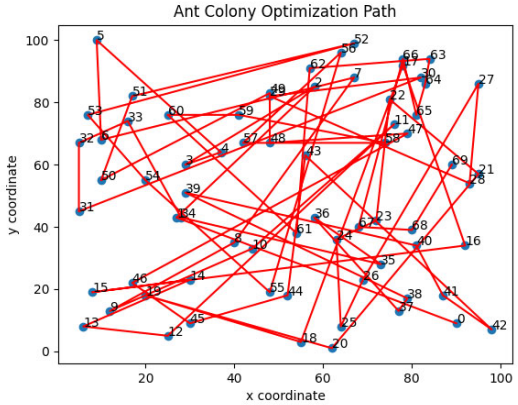
7 实验补充爬山法

7.1 设计介绍

对于Traveling Salesman Problem (TSP,旅行商问题),爬山法是一种较常用的求解算法。
利用爬山法求解TSP的步骤大致如下:
1. 生成一个随机的城市访问顺序作为初始解。
2. 对当前解进行适当的扰动,生成一个新的候选解。扰动的方式可以是交换路径上的两个城市,或者将一段路径反转等。
3. 计算新的候选解的目标函数值(这里是整个路径的长度)。
4. 如果候选解的目标函数值优于当前解,则接受该候选解,反之保持当前解不变。
5. 重复步骤2-4,直到达到终止条件(例如迭代次数耗尽)。
利用爬山法求解TSP的实验分析如下:优点:
- 爬山法实现简单,容易编码。
- 对于小规模的TSP问题,爬山法可以较快地得到较优解。缺点:
- 容易陷入局部最优,不能有效得到全局最优解。
- 对大规模TSP问题效果较差,随问题规模增大,陷入局部最优的可能性增大。
- 终止条件不好设定,迭代次数难以确定。

7.2 结果分析

经过我的实际测试，结果很不理想。可能是迭代次数较少的原因。（结果如图示）



图表 12 爬山法路线图

```
共迭代 500 次，搜索到这种方法的最优解
最优路径： [48, 33, 46, 59, 51, 63, 66, 22, 16, 44, 42, 21, 45, 24, 8, 38, 18, 12, 29, 39, 19, 14, 37, 3, 41, 13, 2, 34, 23,
 57, 30, 61, 47, 55, 20, 31, 5, 27, 7, 11, 6, 25, 54, 52, 43, 26, 60, 62, 4, 36, 53, 64, 35, 10, 32, 67, 0, 9, 65, 49, 50,
 40, 15, 69, 68, 58, 28, 17, 1, 56]
最短距离： 3806.956377350437114
```

图表 13 爬山法程序运行结果

8 实验总结

实验遵循控制变量法,分别实现了贪心算法和蚁群算法,并在同一组TSP问题实例上进行测试对比。

在贪心算法部分,实验采用每次选择距离当前城市最近的未访问城市作为路径的构造策略。该策略贪心地按照局部最优进行扩展,容易陷入局部最优解,无法保证获得整体最优解。

在蚁群算法部分,实验设计了基于问题的信息素启发式函数,引导蚁群搜索。还设计了信息素更新机制,通过积累路径的质量来更新信息素,以此引导蚁群优化搜索方向。

通过实验对比发现,在小规模TSP问题上,贪心算法速度更快,但容易陷入局部最优;而蚁群算法通过集体协作搜索,可以取得更优的近似最优解,但耗时较长。随着问题规模的增加,贪心算法的缺陷更加明显,而蚁群算法表现更为稳定。本实验验证了蚁群算法可以有效求解TSP问题,相比传统贪心算法,具有更好的全局搜索能力和鲁棒性。

实验结果表明,智能算法通过引入随机化机制和积累经验的策略,可以有效帮助算法跳出局部最优,取得更优解。总体来说,本实验通过实例验证和对比分析,检验了了贪心算法和蚁群算法在求解TSP问题上的优劣势,验证了智能算法的有效性,为算法设计提供了借鉴。

完成一个算法实验让我掌握了算法研究的基本流程和方法。从问题分析、算法设计到评测改进,锻炼了我独立研究和总结的能力。这为今后的算法研究奠定了基础。总之,这个实验积累的算法经验和思考方式,将促进我今后算法学习的深入,并更好地应用算法解决实际问题。我会继续努力,进一步提高自己的算法设计与研究能力。