禁忌表算法实验————解决TSP问题

> 本报告使用禁忌表搜素算法解决TSP问题

1. 实验背景

旅行商问题，即TSP问题（Traveling Salesman Problem），也叫旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

旅行推销员问题是图论中最著名的问题之一，即“已给一个n个点的完全图，每条边都有一个长度，求总长度最短的经过每个顶点正好一次的封闭回路”。Edmonds，Cook和Karp等人发现，这批难题有一个值得注意的性质，对其中一个问题存在有效算法时，每个问题都会有有效算法。

迄今为止，这类问题中没有一个找到有效算法。倾向于接受NP完全问题（NP-Complete或NPC）和NP难题（NP-Hard或NPH）不存在有效算法这一猜想，认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

2. 禁忌表搜索算法

2.1 算法简介

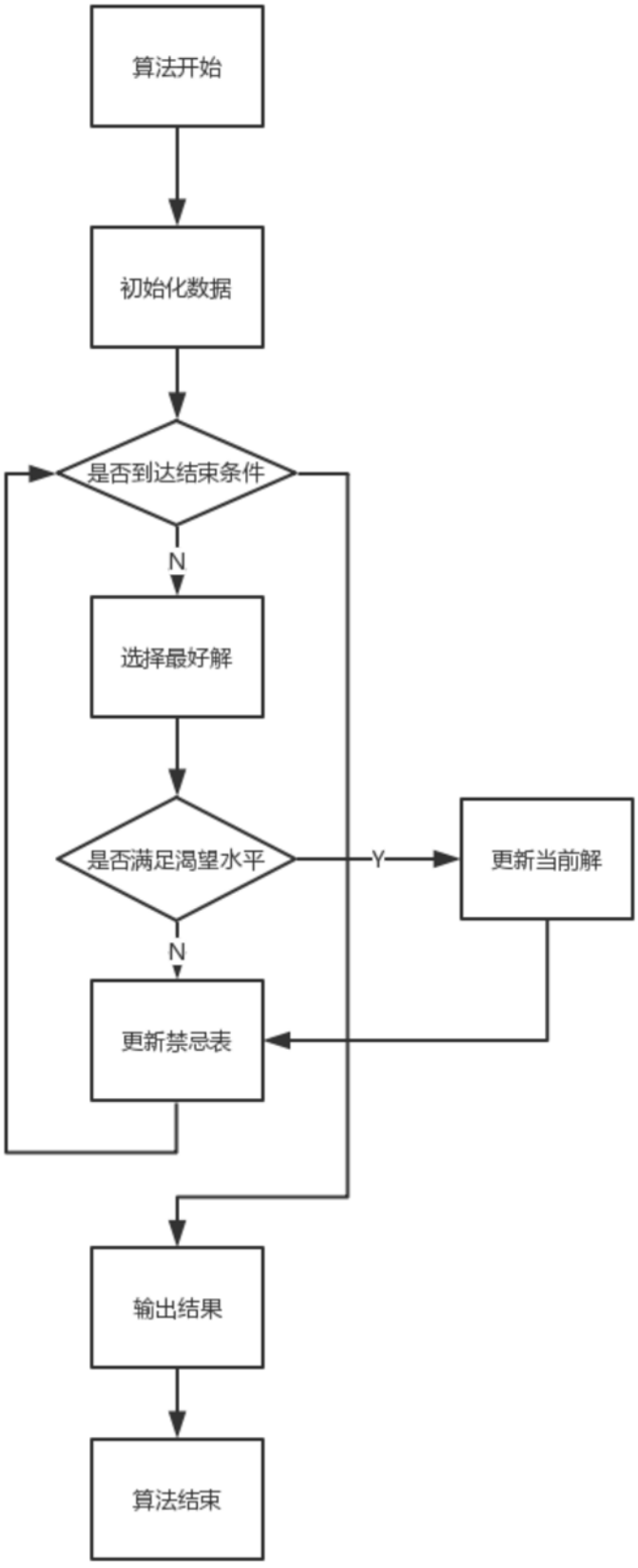
禁忌（Tabu Search）算法是一种元启发式(meta-heuristic)随机搜索算法，它从一个初始可行解出发，选择一系列的特定搜索方向（移动）作为试探，选择实现让特定的目标函数值变化最多的移动。为了避免陷入局部最优解，TS搜索中采用了一种灵活的“记忆”技术，对已经进行的优化过程进行记录和选择，指导下一步的搜索方向，这就是Tabu表的建立。

2.2 算法结合TSP

1. 在搜索中，构造一个短期循环记忆表-禁忌表，禁忌表中存放刚刚进行过的 |T|（T称为禁忌表）个邻居的移动，这种移动即解的简单变化。
2. 禁忌表中的移动称为禁忌移动。对于进入禁忌表中的移动， 在以后的 |T| 次循环内是禁止的，以避免回到原来的解，从而避免陷入循环。|T| 次循环后禁忌解除。
3. 禁忌表是一个循环表，在搜索过程中被循环的修改，使禁忌表始终保持 |T| 个移动。
4. 即使引入了禁忌表，禁忌搜索仍可能出现循环。因此，必须给定停止准则以避免出现循环。当迭代内所发现的最好解无法改进或无法离开它时，算法停止。

3. 算法设计流程

3.1 算法流程图



3.2 算法实现过程

使用python3作为设计语言，其中本次使用的版本是python3.6，其中使用了常规的的库有：

1. time
2. copy

3. random
4. math

使用的非内置库有：

1. numpy
2. matplotlib

3.3 算法设计细节

3.3.1 TSP图类

1. 首先定义点类

```
# 点类
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

2. 随后定义TSP图类，注意 `ask_distance_for_plan()` 的结果

```
class Graph:
    def __init__(self,n):
        self.points = []
        self.point_n = n

    def add_point(self,p):
        self.points.append(p)
        self.point_n += 1

    def ask_distance(self,i,j):
        point_x = self.points[i]
        point_y = self.points[j]
        return math.sqrt(sqr(point_x.x - point_y.x) + sqr(point_x.y - point_y.y))

    def ask_distance_for_plan(self,plan):
        res = 0.0
        for i in range(1,self.point_n):
            res += self.ask_distance(plan[i],plan[i-1])
        res += self.ask_distance(plan[0],plan[self.point_n-1])
        # print(plan,' ',res)
        return res

    def show(self):
        for item in self.points:
            print(item.x," ",item.y)
```

3. 禁忌表的设计
   禁忌表可以使用一个队列，由于Python 语言原生队列比较难以使用，所以自己写了一个类似队列的操作，也就是更新一个新的tabu_table项这个操作：

```
def tuba_step(self,new_plan):
    if new_plan in self.tuba_table:
        return
    elif len(self.tuba_table) < self.MAX_SIZE:
        self.tuba_table.append(new_plan)
    else:
        # 向前推一格
        sz = len(self.tuba_table)
        for i in range(1,sz):
            self.tuba_table[i-1] = self.tuba_table[i]
        self.tuba_table[sz - 1] = new_plan
```

4. 核心部分的设计
   选取适应值作为渴望值，所以禁忌的算法就是：

> - 如果比最优解更优，则更新最优解，并且作为下一个解
> - 否则不断查看下面的解，是否在禁忌表中，直到找到一个不在禁忌表中的更新下一个解
> - 如果没有可以更新，则解禁第一个

所以核心的部分就是：

```
def run(self, n, graph):
    init_plan = self.initPlan(n)
    better_plan = {
        "fitness": graph.ask_distance_for_plan(init_plan),
        "plan": init_plan
    }
    i_time = 0
    while i_time < self.MAX_TIME:
        new_plans = self.get_new_plans(init_plan, 50)
        new_res = []
        for plan in new_plans:
            _fitness = graph.ask_distance_for_plan(plan)
            new_res.append({
                "fitness": _fitness,
                "plan": plan
            })
        new_res.sort(key=cmp)
        # print(new_res)
        # print(new_res,"\n")
        cnt = 0
        flag = False
        while cnt < len(new_plans):
            if new_res[cnt]["fitness"] < better_plan["fitness"]:
                better_plan["fitness"] = new_res[cnt]["fitness"]
                better_plan["plan"] = copy.copy(new_res[cnt]["plan"])

                self.tuba_step(new_res[cnt]["plan"])
                flag = True
                init_plan = copy.copy(new_res[cnt]["plan"])
                break
            else:
                if new_res[cnt]["plan"] in self.tuba_table:
                    cnt += 1
                    continue
                else:
                    flag = True
                    init_plan = copy.copy(new_res[cnt]["plan"])
                    break

        # 如果都在禁忌表中，则解禁第一个
        if not flag:
            if new_res[0]["plan"] in self.tuba_table:
                self.tuba_table.remove(new_res[0]["plan"])
            init_plan = copy.copy(new_res[0]["plan"])

        print(better_plan)

        i_time += 1
    print(better_plan)
    return better_plan
```

5. 其他的细节

   生成初始解

```
    # 初始化一个序列作为初始数据
    def initPlan(self, n):
        random.seed(time.time())
        init_plan = []
        for i in range(0, n):
            init_plan.append(i)

        for i in range(n):
            j = random.randint(0, n - 1)
            while i == j:
                j = random.randint(0, n - 1)
            init_plan[i], init_plan[j] = init_plan[j], init_plan[i]

        # print(init_plan)
        return init_plan

    def tuba_step(self, new_plan):
        if new_plan in self.tuba_table:
            return
        elif len(self.tuba_table) < self.MAX_SIZE:
            self.tuba_table.append(new_plan)
        else:
            sz = len(self.tuba_table)
            for i in range(1, sz):
                self.tuba_table[i - 1] = self.tuba_table[i]
            self.tuba_table[sz - 1] = new_plan

            # 从已有的一个队列中得到一个新的

    def get_new_plan(self, now_plan):
        random.seed(time.time())
        n = len(now_plan)

        i = random.randint(0, n - 1)
        j = random.randint(0, n - 1)
        while i == j:
            j = random.randint(0, n - 1)

        new_plan = copy.copy(now_plan)
        # print("new_plan = ",new_plan)
        # print(i,' ',j)
        new_plan[i], new_plan[j] = new_plan[j], new_plan[i]
        return new_plan

    def get_new_plans(self, now_plan, new_cnt):
        new_plans = []
        for i in range(new_cnt):
            new_plan = self.get_new_plan(now_plan)
            while (new_plan in new_plans) or (new_plan in self.tuba_table):
                new_plan = self.get_new_plan(now_plan)
            new_plans.append(new_plan)
        return new_plans
```

4. 实验结果

4.1 实验数据

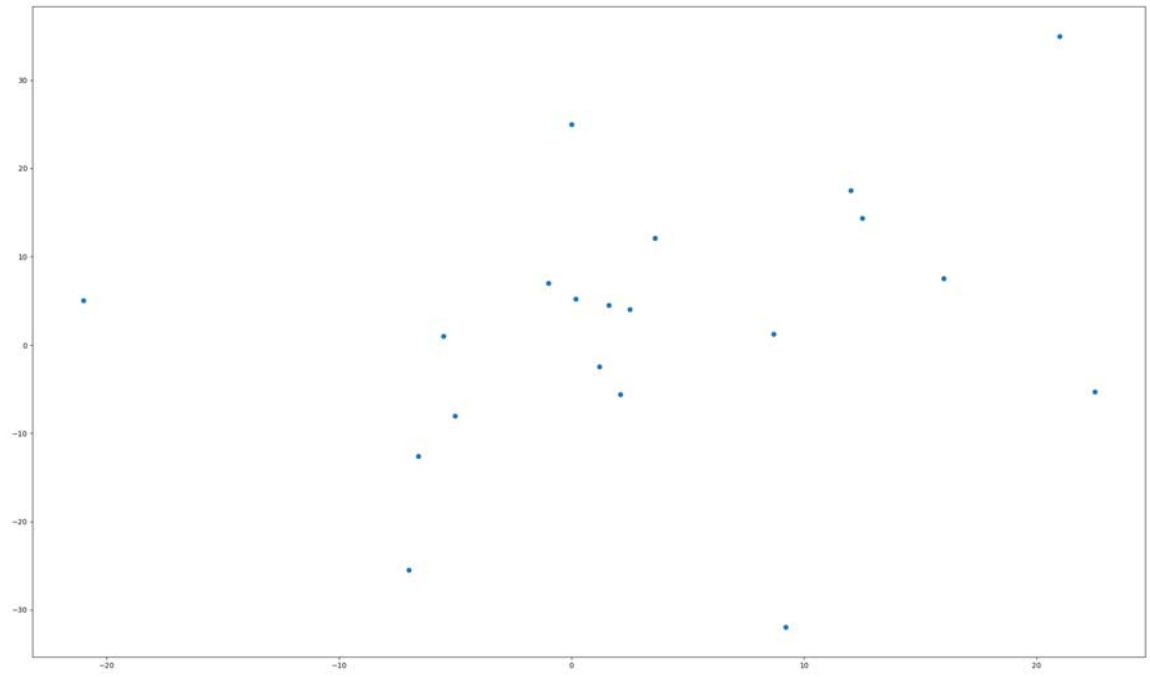
使用的实验数据如下：
给出所有点的坐标

```
A 2.5 4.0
B 1.2 -2.4
C 8.7 1.2
D 3.6 12.1
E -5.5 0.94
F -6.6 -12.6
G 0.18 5.219
H 12.5 14.3609
I 22.5 -5.26
J 1.61 4.5
K 2.1 -5.6
L 0 25
M 9.2 -32
N -1 7
O -5 -8
P 21 35
Q 16 7.5
R -21 5
S -7 -25.5
T 12 17.5
```

给出读入数据的python代码：

```python
graph = Graph(0)
with open("in.txt","r") as f:
    lines = f.readlines()
    for line in lines:
        line = str(line)
        # print(line)
        items = line.split(' ')
        x = float(items[1])
        y = float(items[2])
        # print('x =',x,' y = ',y)
        graph.add_point(Point(x,y))
```

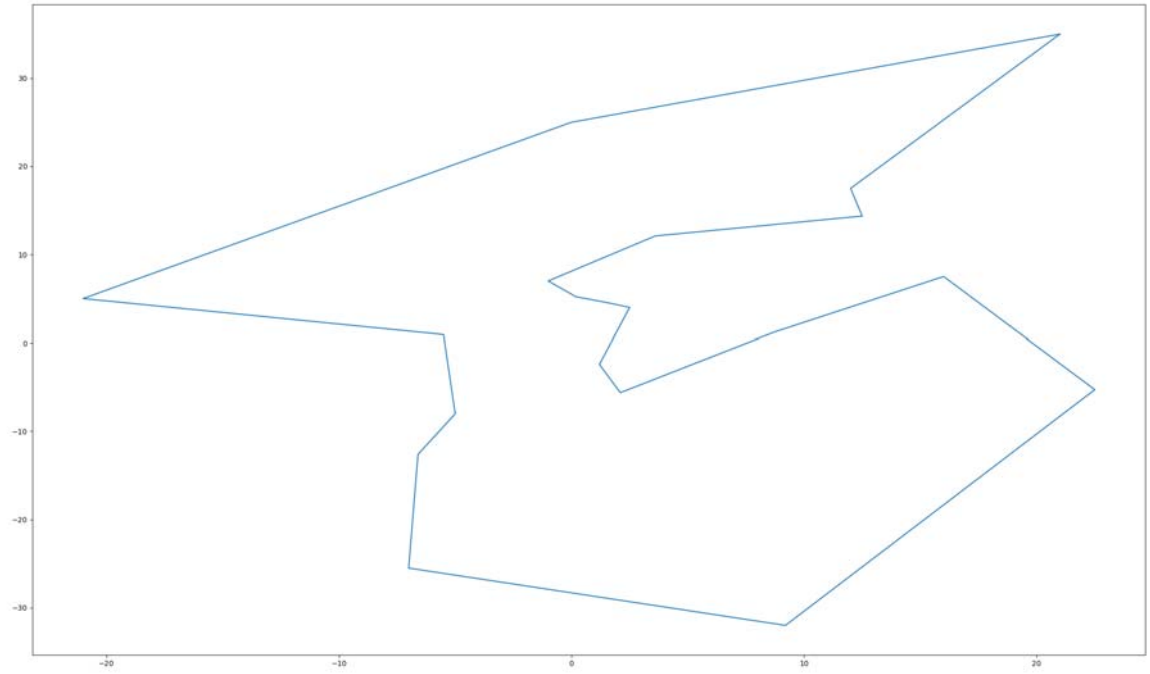
给出一个简单的图：



4.2 实验结果展示

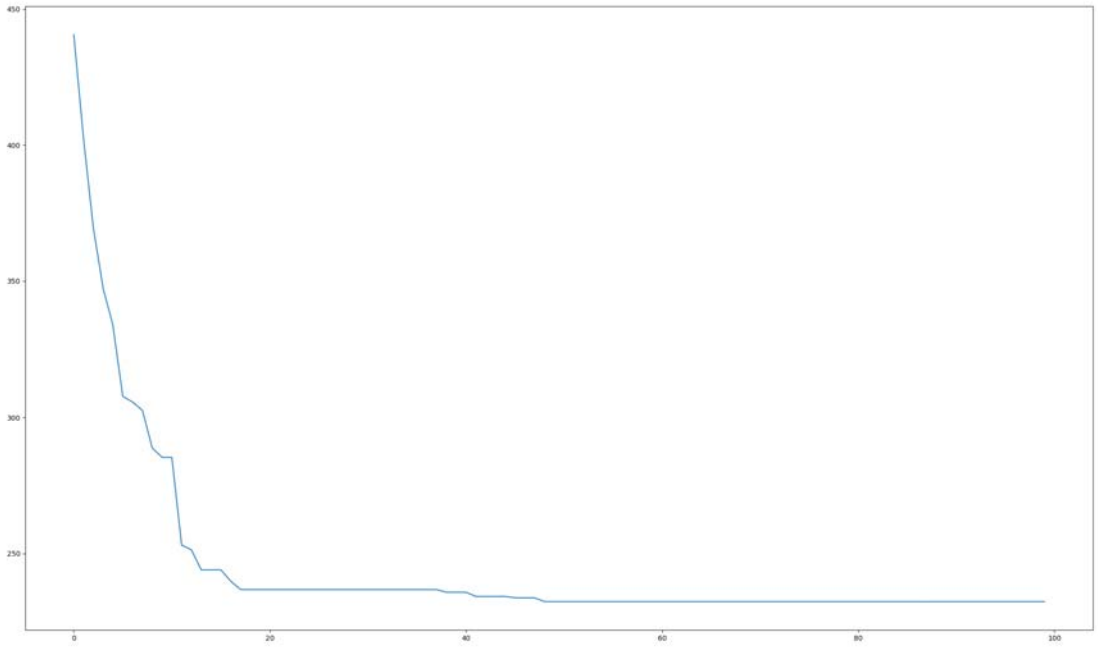
在循环次数在1000，禁忌表的长度为100，每次生成50个解的时候的解:

{'fitness': 229.29301386606764, 'plan': [15, 19, 7, 3, 13, 6, 9, 0, 1, 10, 2, 16, 8, 12, 18, 5, 14, 4, 17

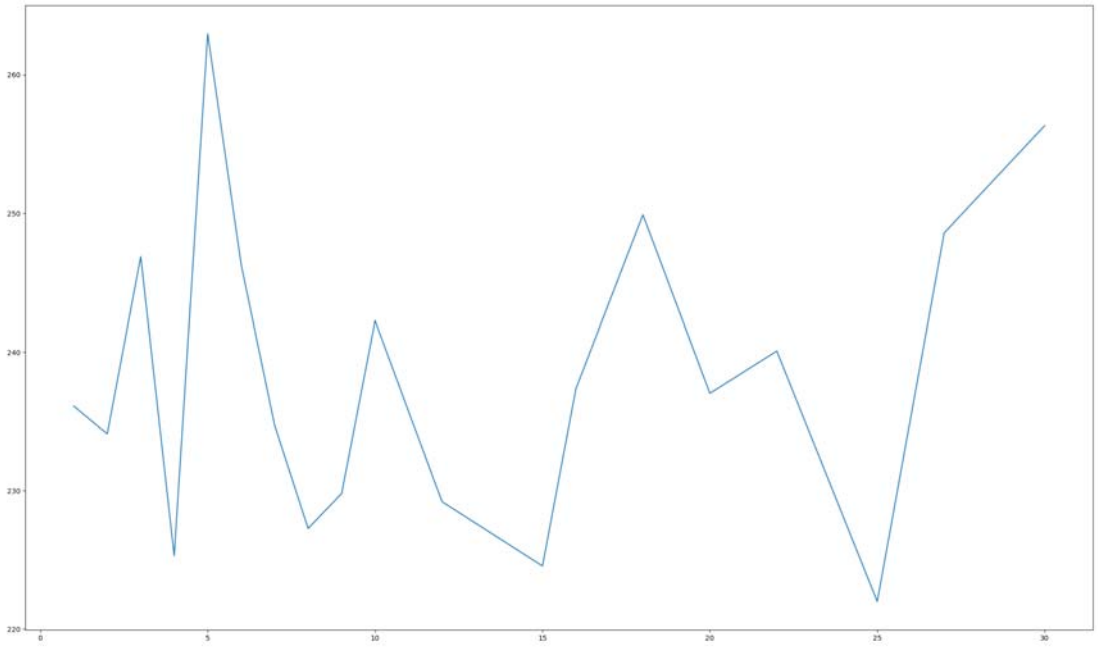


4.2 实验结果分析

1. 修改迭代次数的结果
   查看每次迭代的结果



   可以看到TS搜索是可以非常快速的收敛到最后的结果
2. 修改禁忌表的大小
   对于不同的禁忌表大小



   注意到左边的标签，其实结果还是不错的

```
236.10576449755874
234.09108995597467
246.86027733574613
225.2933368344327
262.9605263450174
246.28820695769377
234.71343544402984
227.26144867324177
229.7833515012846
242.25058271866212
229.1927461840333
224.54844132640252
237.3646981725542
249.88559805941748
237.02818502264515
240.0783888051765
221.98286157579287
248.5770670755478
256.32503827973477
```

可以看到，其实 `tabu_size` 在这个状况下的结果其实还是比较稳定的。