TSP 问题及其求解

开题报告

卢韬
自动化 1903 班
20194127

更新：*May 31, 2022*

摘　　要

TSP 问题已经被研究多年（**?**），已有诸多工作从精确算法（动态规划）、启发式算法（随机贪婪搜索）、亚启发式算法（禁忌搜索（**?**））、元启发式算法（遗传算法、粒子群优化（**?**））等方向入手，在获取精确解的能力上取得了较好的效果。然而，由于 TSP 问题 NP 难的特点，没有单一的方法能快速、精确地求解各种类型的 TSP 样例。而这些算法评价指标之间普遍存在着诸多 trade-off，单一的求解最优解的能力不足以衡量算法好坏。因此面对不同特点的算例样本，需要一种科学的评价指标衡量选取哪种算法更好。本项目将讨论多种用于求解 TSP 问题算法的原理，并将使用 python 复现、对比各类算法。最终，结合实验过程和关键评价指标（精度、复杂度、速度）给出一种统一的算法评价指标计算方法，并得到不同特点的算例下最适合的算法。所有的代码和笔记将开源在：github 开源地址。

关键词： TSP 问题，遗传算法，禁忌搜索，贪婪算法，动态规划算法，粒子群优化算法

目录

1 TSP 问题介绍

旅行推销员问题（Travelling salesman problem, TSP）描述如下：给定一系列城市和每对城市之间的距离，求解访问每一座城市一次并回到起始城市的最短回路。它是组合优化中的一个 NP 难问题，在运筹学和理论计算机科学中非常重要。TSP 问题在 1930 年代在维也纳和哈佛大学首次开始研究，在 1950 年代和 1960 年代在学术界开流行，当时的最优模型可以解决 48 城市的 TSP 问题。在 1970 年代，解决能力达到 120 维，不久又达到 318 维。如今，目前的最强模型能求解多达 24978 维的 TSP 问题。从图论的角度来看，该问题实质是在一个带权完全无向图中，找一个权值最小的 Hamilton 回路[1]。由于该问题的可行解是所有顶点的全排列，随着顶点数的增加，会产生组合爆炸。1972 年，Richard Karp 发表了一篇 17k 引用的论文**?**，证明了哈密顿循环问题是 NP 完全的。这意味着数学上来说，旅行商问题是 NP 难的。

TSP 问题具有实际应用背景。目前被广泛应用于公共服务可靠性、信号控制、路径规划（**?**）、PCB 生产（**?**）、流水车间问题及其变体（**?**）等诸多实际问题中。

1.1 TSP 问题的数学模型

对 TSP 问题有如下数学描述：

$$\min \sum_{i,j=1}^{n} x_{ij} d_{ij} \tag{1}$$

Subject to：

$$\sum_{i \in V} x_{ij} = 1, \forall i \in V_c \tag{2}$$

$$\sum_{j \in V} x_{ij} = 1, \forall j \in V_c \tag{3}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \forall S \subset V_c, 2 \leq |S| \leq n \tag{4}$$

公式中未知数的含义如表**??**：

表 1: 未知数定义

未知数	含义	取值范围	注释
d_{ij}	i 到 j 的距离	$[0, -\infty)$	距离衡量可以用 p 范数
x_{ij}	在 i 后访问 j	0,1	使用 one-hot 编码
i , j	城市代号	$i \neq j$	不超过城市数量
N	节点集合	len(N)=n	所有城市的集合

[1]哈密顿图是一类无向图，从指定的起点前往指定的终点，途中经过所有其他节点且只经过一次。闭合的哈密顿路径称作哈密顿回路。

1.2 TSP 问题数据集

TSP 问题可用的数据集非常之多，比如**?** 提出的 TSPLIB—A、

根据 Google 公司的一篇文档[2]对一般 TSP 问题测试样例数据的规约，可以得到读取数据集方式，具体读取代码在附录中。

附录中给出了快速下载 TSP 测试样例[3]的下载 python 脚本。

附录中还给出了提取文件夹中所有.tsp 样例的 python 代码。

可以使用G.remove_edges_from(nx.selfloop_edges(G))来去除自环。利用图论可视化工具（**?**），可以将邻接矩阵读取并转化为邻接关系数列，并进一步可视化无向图如图**??**。

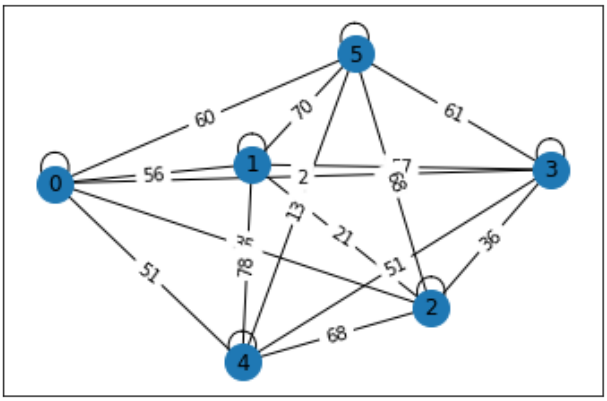


图 1: TSP 数据可视化（蓝色边为可行驶路径。）

1.3 ATSP 问题转化为 TSP 问题

在上述的 TSP 问题中，算例给出的通常是城市的二维坐标。i，j 两点之间的距离 $d_{i,j}$ 通过欧拉距离（2-范数）计算，因此两点间不同向的边距离是相同的。此类图组成的邻接矩阵有对称性，在图论中形成一个无向图，而这种天然的对称性将解的数量减少了一半。但实际生产过程中，部分问题只能转化为非对称旅行商问题（Asymmetric Travelling Salesman Problem, ATSP），其中 $d_{i,j} \neq d_{j,i}$。此类图组成的邻接矩阵无对称性，在图论上可转化为有向图。比如在《现代优化计算方法》课程的实验一中，需要解决的问题就是 ATSP。

将未知类似问题转换为特殊情况下的已知问题是常用的手段，因此将 ATSP 转换为 TSP 问题求解将有助于扩大 TSP 求解能力。因此根据其特点，有如下两种方式将 ATSP 问题转换为 TSP 问题求解。

[2]数据详细解释下载地址

[3]下载源来源：数据集下载地址

1.3.1 改写距离获取函数

由于 TSP 问题给出的邻接矩阵具有对称性，因此 TSP 算法在设计时就会利用对称性去重[4]。这样就导致以下三个问题：

- .atsp 和.tsp 测试样例保存的格式不同。
- TSP 算法中对称邻接矩阵保存的数据结构会降重，导致缺失非对称距离储存能力。
- TSP 算法中对称邻接矩阵距离读取函数不具备非对称读取能力。

因此想要将 TSP 算法的求解程序应用到 ATSP 问题上，一种直接的思路是利用 python 的重写特性，以上三个位置的函数重写即可。对于启发式算法来说，这样的转化是非常完美的。

1.3.2 构造新的邻接矩阵

经过文献检索，我找到了一种将非对称 TSP 问题转换为对称 TSP 问题的方法（**?**）。通过这种方法，我们可以将非对称 TSP 问题转化为对称 TSP 问题，然后使用适合对称问题的算法求解该问题，而不是重新设计算法。

此算法将一个 A 称 TSP 问题的距离矩阵 $\boldsymbol{C}$ 转化为对称 TSP 问题的距离矩阵 $\tilde{\boldsymbol{C}}$。这样，只需要在读取数据是进行相应转换，就可以无须改动其他代码而直接使用已有 TSP 求解方法。算法步骤如下：

1. 令 $\overline{\boldsymbol{C}} = \boldsymbol{C}$，其中 $\overline{c_{ij}} = -M \quad (i \in N)$。其中 M 为相当大的数，表示无法通行。
2. 取 $\boldsymbol{U_{n\times n}}$ 为 n 维方阵。即 $i, j \in N$，有 $u_{ij} = \infty$。
3. 有 ATSP 问题的转化对称距离矩阵

$$\tilde{\boldsymbol{C}} = \begin{pmatrix} \boldsymbol{U} & \overline{\boldsymbol{C}}^{\mathrm{T}} \\ \overline{\boldsymbol{C}} & \boldsymbol{U} \end{pmatrix} \tag{5}$$

4. 此时新的城市节点集合为：

$$\tilde{N} = \{1, 2, \ldots, n, n+1, \ldots, 2n\} \tag{6}$$

5. 此时需要转化后 TSP 问题得到的最优解有以下以下形式：

$$N(i_1) \to N(i_1+n) \to N(i_2) \to N(i_2+n) \to \ldots \to N(i_n) \to N(i_n+n) \to N(i_1) \tag{7}$$

其对应的转化函数如下，拼接过程使用了 numpy 库。

```
# 为了使算法方便观察，没有使用节省内存和更快的写法，
# 但减少中间变量和新建数组，会显著加快程序运行
def ATSP2TSP_np(matrix):
    lu = np.ones_like(matrix)*(10000)
    ru = matrix.T
    rd = np.ones_like(matrix)*(10000)
```

[4]参考对上三角矩阵的优化储存。

```
    # 先竖着拼接
    left = np.concatenate((lu,matrix),axis=0)
    right = np.concatenate((ru,rd),axis=0)
    # 整体拼接
    return np.concatenate((left, right),axis=1)
```

[

2 求解 TSP 问题的算法

根据所学内容，TSP 问题有有如图**??**常用算法。总体分为精确算法、启发式算法和深度学习算法。

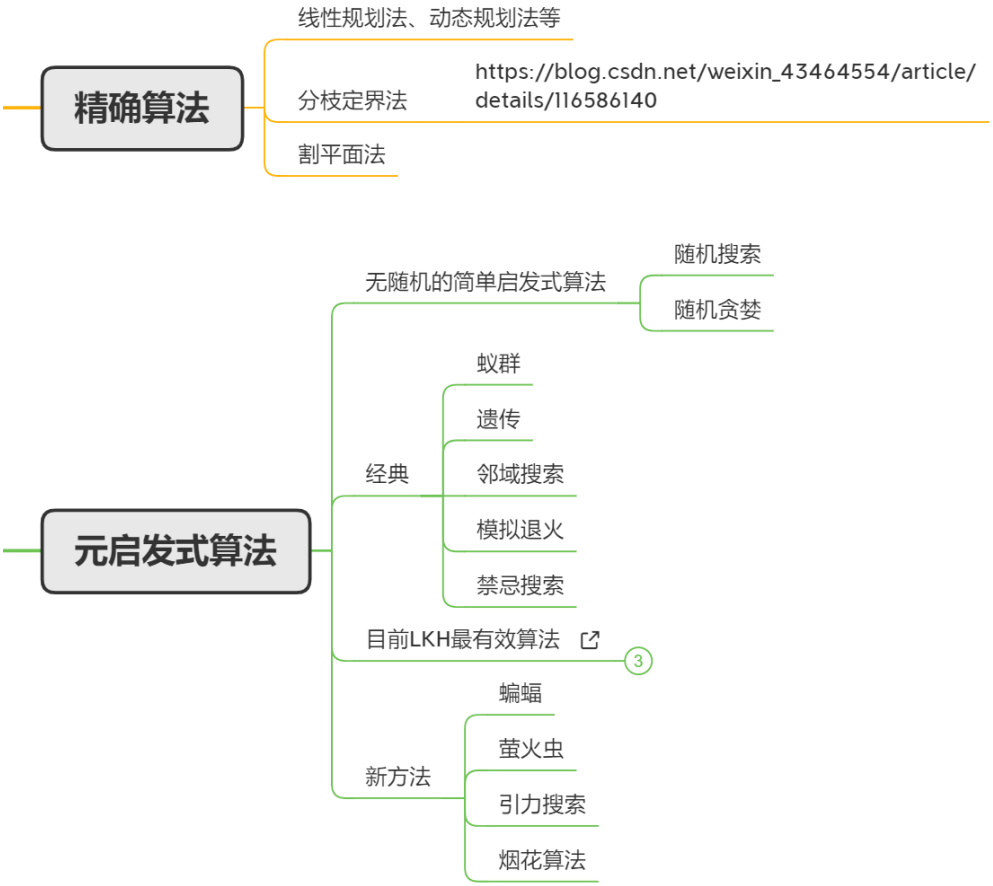


图 2: 算法汇总

2.1 精确算法

最直接的求解算法是对所有可能的路径进行完整枚举，以确定成本最低的路径。然而，对于 n 个城市求解的时间复杂度为 $O(n!)$ 时间，这种方法仅适用于较小的城市规模。

2.1.1 分支定界法

根据 2021 秋季学期课程《系统优化与人工智能》和《系统工程概论》，分支定界方法常用于求解各类优化问题。

ILP 首先被宽松并使用 Simplex 方法作为 LP 求解，然后通过枚举整数变量重新获得可行解。

2.1.2 线性规划算法

2.1.3 动态规划算法

2.2 启发式算法

NP-hard 问题难以在大尺度上以最优方式精确求最优解。启发式算法常用于给出一个好的近似解，但不一定是最优解。这些算法不保证最优解，但在合理的计算时间内给出接近最优的解。

2.3 深度学习算法

基于深度学习的算法准确率已经接近经典方法。但目前效率仍然较差**??**，在数百节点左右的规模上就很难快速收敛。因此无法将学习到的策略推广到实际规模的更大实例。**?** 的工作将目前主流的深度学习技巧重新整合，并提出了合适的网络结构和学习方式。同时其开源代码已有较多使用量，非常成熟。

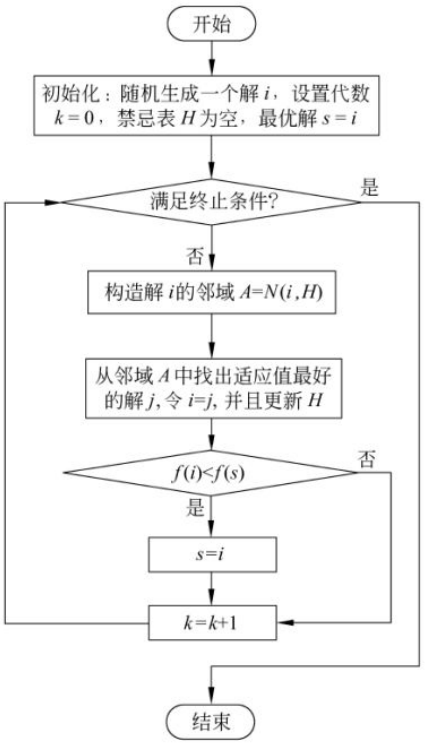
表 2: 在不同的 TSP 大小和搜索设置中，SL 和 RL 训练的模型的近似训练时间（12.8M 样本）和推理时间（1280 样本）。GS: 贪婪策略，BS128: 宽度为 128 的束搜索，S128: 采样 128 个解决方案。RL 训练使用推出的基线，计时包括每 128,000 个样本后更新基线的时间。

图尺寸	训练时间[5]		推理时间		
	SL	**RL**	**GS**	**BS128**	**S128**
TSP20	4h 24m	8h 02m	2.62s	7.06s	63.37s
TSP20-50	9h 49m	15h 47m	-	-	-
TSP50	16h 11m	40h 29m	7.45s	29.09s	86.48s
TSP100	68h 34m	108h 30m	19.04s	98.26s	180.30s
TSP200	-	495h 55m	54.88s	372.09s	479.37s

3 禁忌搜索算法

3.1 禁忌搜索算法求解 TSP 问题

禁忌搜索（Tabular Search，TS）最初由**?** 提出。禁忌搜索是一种亚启发式随机搜索算法，它从一个初始可行解出发，通过随机指定的变化规则（比如 2-gram），选择实现让特定的目标函数值变化最多的移动。本质上是通过扩大邻域搜索结构来得到更好的解。为了避免陷入局部最优解，TS 搜索中采用了禁忌表作为记忆机制。对已经试探过得较优解，能在一定时间内避免尝试从而避免震荡求解的情况。




```
//功能：禁忌搜索算法伪代码
//说明：本例以求问题最小值为目标
//参数：i 为当前解；s 为最优解；H 为禁忌表；k
//      为当前代数
procedure TS
  //Initialization
  Randomly generate a solution i, and
  evaluate its fitness f(i);
  s=i;k=0;H={};
  while not stop
      //Generate the neighbors of solution i
      //based on the tabu list H
      A=N(i,H);
      i=SelectBestSolution(A);
      Update the tabu list H;
      if f(i)<f(s)
        s=i;
      end if
      k=k+1;
  end while
  print s
end procedure
```


图 3: 禁忌搜搜算法求解 TSP 问题思路

根据算法介绍，可以得到流程框图，如图 **??**。

3.1.1 算法关键要素

1. 禁忌对象的选取 (禁忌表中被禁的那些变化元素)
   - 解的简单变化：将原来的解和变化后的解都记录下来。如：$ABCDE \rightarrow ACBDE$
   - 向量的变化：记录变化位置和变化量。如：$(1,1,0,1,0,1) \rightarrow (1,0,0,1,0,1)$
   - 目标值的变化: 禁止相同目标值的解被接受。如：$f(x)$。
2. 邻域结构（决定邻域按什么样的方式生成）
   - 使用 k-opt，既交换当前解级的 k 个城市。(123456) → (126543)

- 比如 **?** 中使用的 2-opt 方法，每次交换两个城市，如 (123456) → (123465)。

3. 禁忌长度的确定 (被禁对象不允许选取的迭代代数，tabu(x) = t)
    - t 为给定常数。
    - t 为某一范围。
    - t 动态变化，比如我使用的余弦下降、幂指数下降等策略。

4. 特赦规则 (aspiration criteria)
    - 基于评价值的规则：遇到更好的解。
    - 基于最小错误的规则：当所有对象都被禁忌时，选择最好。
    - 基于影响力的规则：对目标函数的影响大小。

5. 候选集合的确定
    - 选择邻域中目标值最佳的若干个邻居。
    - 随机选取。

6. 终止规则
    - 确定步数终止：迭代代数超过给定值 K，终止计算。
    - 频率控制原则：某一解、目标值、元素序列的频率超过给定标准，终止计算。
    - 目标控制原则：给定步数内目标值没有改进。
    - 目标值偏离程度原则：$f(x) - Z_{LB} \leq \epsilon$。

3.1.2 算法实现

根据如图**??**算法探究，我完成了使用禁忌搜索的 TSP 求解。这里我选择了 ATSP 作为测试样例，但是本质上与 TSP 都是相通的。此类问题有两个思路，转换为 TSP 或者按照 ATSP 的非对称路径来读取。我将两种思路都实现了，最终对比发现后者在储存、速度上都更好。

3.1.3 算法优化

本模型首次完成时间较早，但是算法性能始终不好，因此反复尝试了一些技巧来提升性能。从最初的 2500+（最好）到现在的 1900+（最好）。本次实验的代码完全由本人完成，没有借鉴任何代码。

由于算法从零开始实现，因此数据结构的选取、迭代过程中中间数据的处理等细节有诸多实现方法。因此在完成实验内容后，我利用 python 中的修饰器技巧，通过函数包装来快速获取各个函数的运行时间，便于找到耗时的瓶颈位置。只需要在写好的函数上加入 @timmer 即可在调用过程中自动记录时间。此方法来源于我的个人博客，在初学 python 时候记录的笔记。

```
import time
def timmer(func):     #传入的参数是一个函数
    def deco(*args, **kwargs): #本应传入运行函数的各种参数
        print('\n函数：{_funcname_}开始运行：'.format(_funcname_=func.__name__))
        start_time = time.time()#调用代运行的函数，并将各种原本的参数传入
```

```
        res = func(*args, **kwargs)
        end_time = time.time()
        print('函数:{_funcname_}运行了 {_time_}秒'
              .format(_funcname_=func.__name__, _time_=(end_time - start_time)))
        return res#返回值为函数
    return deco
```

[

最终在算法运行速度优化上有总体 85% 时间优化。比如在读取距离方法、ATSP 问题处理等多个算法耗时的瓶颈处做了消融实验，选取了运行速度更快的解决方案。同时在算法书写时也使用 lamda 表达式排序等方法尽量减少代码复杂度。在执行时间判断函数过程中，使用了 jupyter notebook 独有的魔术指令\%\%!，在每个单元格中都运行多次同一份代码，求取平均值作为最终结果。下图记录的是数据结构的决定。

3.1.4 训练过程参数控制

为了保证算法的泛化性能，在代码实现过程中将所有可控变量解耦合。用较多的封装保证算法可以很容易的进行参数调节。在 run () 函数中，可以通过改变超参数传入，容易的完成各项功能的消融实验。下表所示的是算法的超参数单独单元定义。

```
# 算法需要的变量
tabLength = 50                      # 禁忌表长
banned_Table = []                   # 禁忌表
city_Num = tab_np.shape[0]          # 城市数
cityNum = tab_np.shape[0]
randSeed = 42                       # 随机数种子
random.seed(randSeed)               # 给random库设计随机数生成器
maxIter = 1000                      # 终止准则
neighborNum = 20000                 # 生成的邻居数量
solution = list(range(city_Num))    # 解
solution_list = []                  # 生成多个解构成的数组
exchange_Num  = 2                   # 领域交换量 2-opt
# 当前路径长度（加一个变化的权重，路径长度和value不一样，非线性,比如平方）
# 加大对离谱解的惩罚力度，但要小心震荡
thisLength = 1000000
thisValue = 1000000
random_accept= 0.05                 # 随机接受概率
```

[

同时我也结合自己在深度学习中学习的技巧，将 lerning rate（tabu 长度）下降、抖动（长时间梯度不下降就跳开）、允许随机犯错等手段加入了算法实现过程中，尽可能优化算法。

观察到表长度较小的时候，很容易出现算求解来回震荡的情况。因此算法需要在表长衰减时候做截断，来保证搜索能力。

4 粒子群优化算法

4.1 粒子群算法介绍

根据 PSO 课程上记录的笔记（图**??**），可以得到算法的流程图和重要参数。

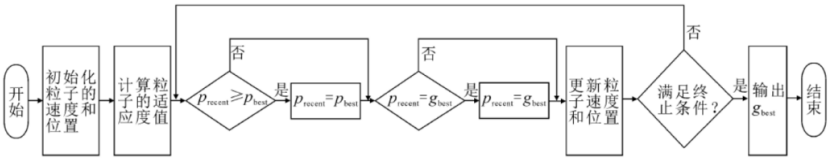


图 4: 粒子群算法标准流程

4.2 超参数（粒子个数、迭代次数等）对搜索性能的探究

由于 PSO 算法性能很大程度上取决于超参数选取，因此想要达到更好的性能，需要对超参数进行详细对照实验。但是使用整体算法求解 TSP 问题时速度较慢，很长时间才能得到最后的解，不利于调参。所以这里先选用几个常用的测试函数对算法进行大致探索研究，再将其迁移到现有问题上。

这里选用 Griewark 函数、Rastrigrin 函数和 Rosenbrock 函数三种典型测试函数。根据实验结果和 TSP 问题特性来做具体参数选择。

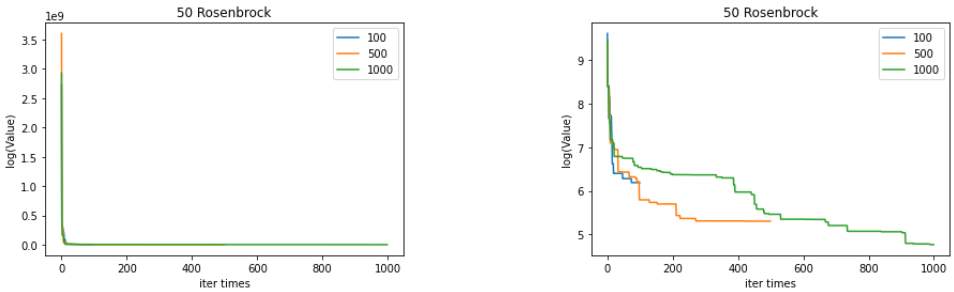
观察 PSO 算法在粒子个数分别设置为 50、100 和 300，迭代次数分别设置为 100、500 和 1000 情况下的寻找最优解的过程。

4.2.1 使用对数坐标系进行可视化

当使用 30 维的多变数扩展测试函数时，算法搜索范围可以达到（1e0,1e9），跨度很大。如下图**??**（a）所示，可以观察到三种颜色表示的三种参数下的三条曲线紧贴在一起，无法直观观察的算法的搜索过程。

因此，考虑到模电和自控原理中 bode 图的对数坐标系，我将数据进行了对数化，使用纵轴为对数坐标系的方法，使得不同参数情况下算法的可视化可以直观体现出来（如图**??**（b））。具体实现时，可以采用 matplotlib 的对数系，也可以采用 numpy 将数据整体转换，还可以手工转换。这里为了更好体现算法细节，使用手写转换方式。

观察到三条曲线成功分离开了，便于算法性能的对比研究。但是也要注意，对尺度进行缩放也导致差距较为接近的两条线水平方向重合，其斜率也被去了对数，变化趋势不敏感。因此对于算法执行末期，数量级差距不大情况下，还应该回到标准坐标系下探究。



(a) (b)

图 5: 坐标变换效果对比

4.3 算法实现

同时，吸取前两次实验中算法实现使用顺序结构的弊端。设置过多全局变量定义的超参数导致进行重复实验时出现变量泄露等问题，这些问题在 python 中（尤其是 jupyter notebook 中）debug 很困难。此次算法实现用类来封装，超参数使用类内变量，使得创建和销毁都更安全，做多组对照实验时出现问题较少。

同时，由于整体算法在求解 TSP 问题时速度较慢，不利于调参。这里先选用几个常用的测试函数对算法进行大致探索研究，再将其迁移到现有问题上。

这里选用 Griewark 函数、Rastrigrin 函数和 Rosenbrock 函数三种典型测试函数。

4.4 对照试验

4.4.1 实验设计

考虑到实验中涉及到三组变量：目标函数、迭代次数、种群量。为了直观展现这些变量之间的关系，我使用了 matplotlib 的子图绘制和隐函数的迭代器，可以方便的进行重复对照实验。同时，我也使用了变量控制，每个子图中的三条下降曲线都只有一个超参数变化。比如：当使用 50 种群搜索 rosenbrock 函数时，迭代次数分别设置为 100、500、1000 进行试验，其图像放置于大图**??**的（1，1）位置。

图中：表头为次小图中实验的两个固定参数，用于固定此 2 参数，比对图例中的参数。大图中的 9 个小图表示不同参数情况下的运行情况，用于比对其余两参数的情况。

4.4.2 实验分析

从图**??**可以看到不论对于何种函数或采用何种规模种群，其中总存在非常大的“平台期”[6]，也存在有平台期对应的拐点。比如观察到对于 Rastrigrin 和 Griewark 函数，在 250iter 后搜索的边际效益逐渐降低，种群已经逐步落入局部最优的陷阱，很难再有进步。继续增加 iter，算法几乎没有进步。而对于 Rosenbrock 函数，由于其搜索难度较大，因此更多的迭代次数能显著增加结果的最优

[6]既多个迭代次数后算法达到的最优解没有更新。

性。根据 100 次的重复实验，也可以判断对 Rosenbrock 函数，其搜索结果与迭代次数具有较强线性相关性。

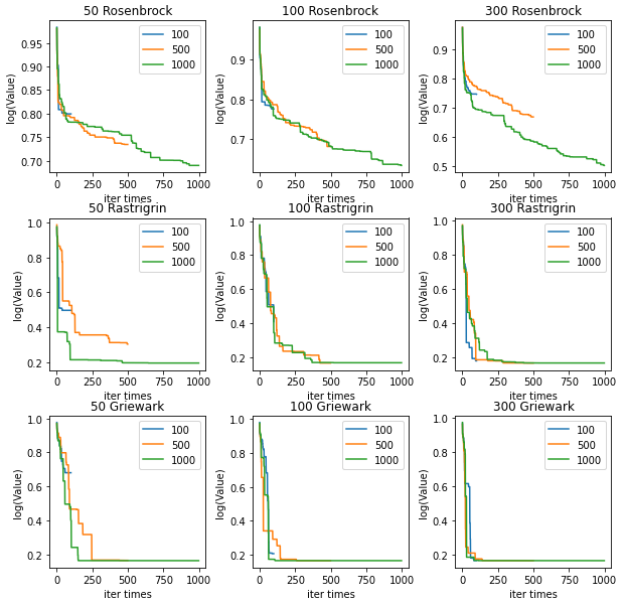


图 6: 对迭代次数的探究

通过衰减速率来看（需要注意的是，图示衰减速率取了对数，但不影响大小关系），大的种群能有更持久的搜索能力，图**??**（2，1）、图**??**（2，3）可以明显看出较小的种群容易陷入某个局部最优值，而种群数量大就有更多机会持续找到更优的局部最优值。既对于大多数情况，大种群的收敛位置要由于小种群。这样的想法也在更大规模、更细密的采样上得到了实验。这里推荐使用 google 的 colab pro 平台，其提供的服务器使用多个 Intel(R) Xeon(R) CPU @ 2.20GHz，运行速度要比本地快很多，支持了本次大规模测试。我也共享了我的代码，连接在最前方。

但是同时，我通过做额外的对照试验（图**??**）发现，过大的种群（5000）反而会导致收敛非常慢，计算时间也呈指数倍增长。而且根据问题不同，搜索算法也不是一定会明显更好。种群规模与算法效率、算法性能之间存在 trade-off，需要在具体实现测试样例中分析合适的参数取值。比如在 griewark 测试样例中，算法性能就没有明显提升。

这里运算计算时间利用 python 的修饰器完成，既将计算函数传入 @ 开头的另一个修饰器函数中。

也可以使用 tqdm 的迭代器来完成（如图**??**），还可以用 python 自带的分析测试工具完成。观察到计算时间随着种群数量指数级上升，图中也展示了多次运行求平均（减少随机性干扰）的思想。需要注意的是，python 中可以使用魔术指令%timeit（或者 cProfile 分析）来自动重复运行单元格，获取平均算法耗时，但是多次运行后，部分变量会被系统优化进入 L2chahe 缓存、或者被其他解释器自带优化方法优化，其统计的平均耗时将会小于真实情况。因此手写重复运算可能是更合适的方法，这样内循环的变量会被 python 自动回收，不会被机器优化。

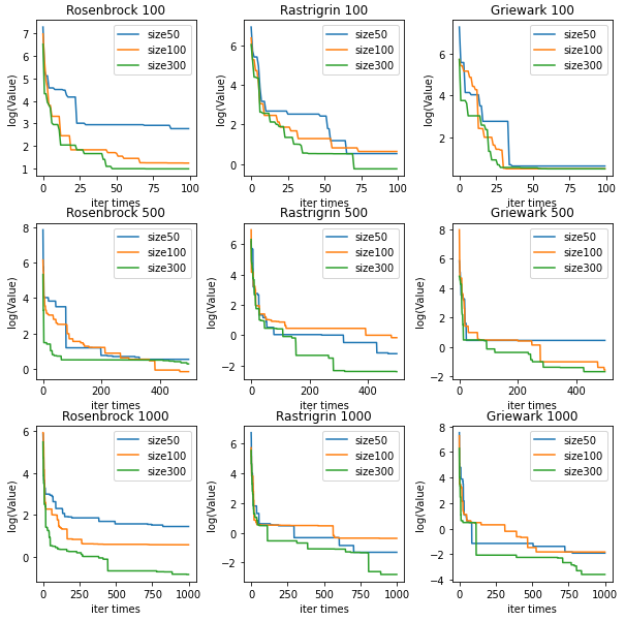


图 7: 对种群大小的探究

5 蚁群算法

6 总体算法实现、代码亮点

6.1 可复现性

随机性运行环境：线上平台、requirement.txt

6.2 可视化

如图**??**，本项目使用了 matplotlib 中的可视化方法，并加以封装。同时通过训练过程中数据结构保存中间结果，便于直观感受算法运行情况，给出调参数的方向。

使用了 tqdm 库，来可视化算法的训练进程。在大规模测试情况下得以判断运行时间。

6.3 TSP 问题路径可视化

使用复杂网络处理工具 networkx[7]（**?**）有助于直观展示算法效果。比如环的散乱程度（图**??**）。此方法将numpy.ndarray格式的临界矩阵数据经过方阵检查、编号、去重复等处理转化为邻接表，进而可视化。

[7]复杂网络工具 networkx

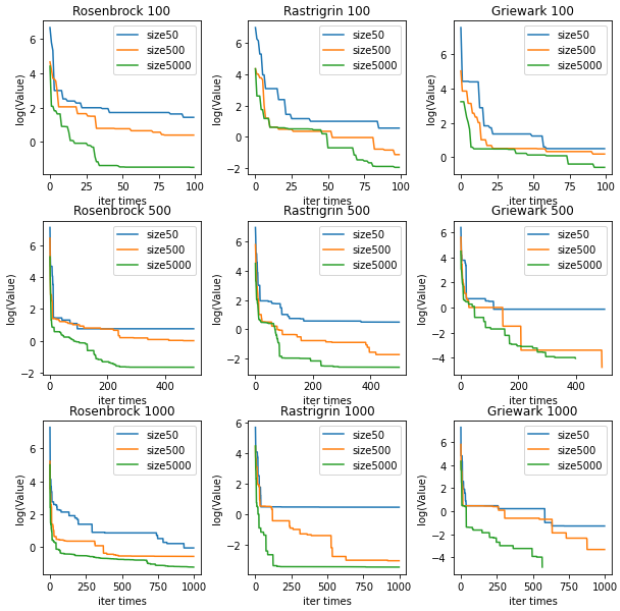


图 8: 种群量额外的对照实验（种群量为 50、500、5000）

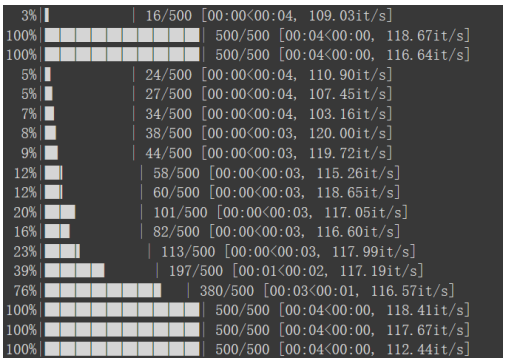


图 9: 不同种群量计算耗时分析（单位：每秒多少次迭代）

6.4 调试工具

6.4.1 算法效率

6.4.2 算法可靠性

使用 warning、assert 等错误处理机制，便于切换测试样例带来问题的排查。

Calculate pheromone trails limits: $\tau_{\min}$ and $\tau_{\max}$
Set pheromone trails values to $\tau_{\max}$
$global\_best \leftarrow \emptyset$
for $i \leftarrow 1$ **to** *#iterations* **do**
    for $j \leftarrow 0$ **to** *#ants* $- 1$ **do**
        $u \leftarrow \mathcal{U}\{0, n-1\}$ `// Select the first node randomly`
        $route_{\text{Ant}(j)}[0] \leftarrow u$
        Add u to $tabu_{\text{Ant}(j)}$
        for $k \leftarrow 1$ **to** $n - 1$ **do** `// Complete the solution (route)`
            $u \leftarrow$ select_next_node$(route_{\text{Ant}(j)}[k-1], tabu_{\text{Ant}(j)})$
            $route_{\text{Ant}(j)}[k] \leftarrow u$
            Add u to $tabu_{\text{Ant}(j)}$
    $iter\_best \leftarrow$ select_shortest $\left(route_{\text{Ant}(0)}, \ldots, route_{\text{Ant}(\#ants-1)}\right)$
    if $global\_best = \emptyset$ **or** $iter\_best$ is shorter than $global\_best$ **then**
        $global\_best \leftarrow iter\_best$
        Update pheromone trails limits $\tau_{\min}$ and $\tau_{\max}$ using $global\_best$
    Evaporate pheromone according to ρ parameter
    Deposit pheromone based on $iter\_best$

图 10: The MAX-MIN Ant System.

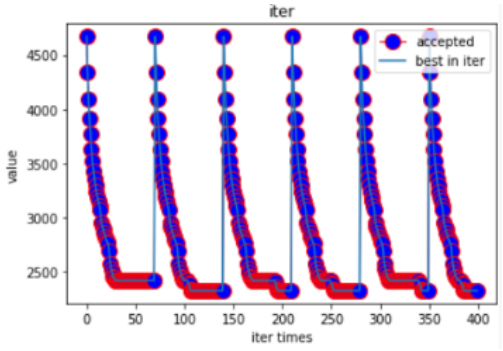


图 11: 同一个初始解进行多次搜索

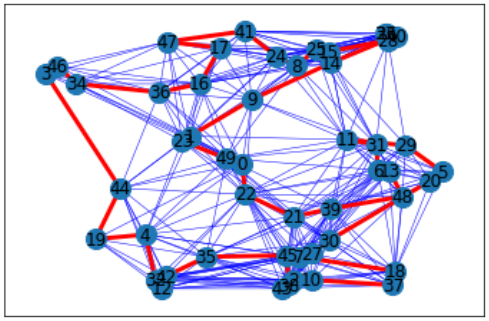


图 12: TSP 路径可视化（其中红色边为最终结果，蓝色边为可行驶路径。）

A 附录

附录中存放过长的代码和尺寸过大的图片。

A.1 程序代码

```
# 获取数据集
import requests,os
from lxml import etree
from urllib.parse import urlparse
url = 'http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html'
resp = requests.get(url)
text = resp.text
html = etree.HTML(text)
parse = urlparse(url)
root = 'http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/'
for a in html.xpath('//li/a'):
    name = a.xpath('./@href')[0]
    path = os.path.join('res', name)
    if os.path.exists(path):
        print('已存在%s' % name)
        continue
    durl = root + name
    dresp = requests.get(durl)
    with open(path, 'w') as fp:
        fp.write(dresp.text)
    print(name)
```

[

```
# 读取文件夹下所有.tsp文件
import os
import pandas as pd
path = "./" #文件夹目录
files= os.listdir(path) #得到文件夹下的所有文件名称
s = []
for file in files: #遍历文件夹
     if not os.path.isdir(file) and file[-4:-1]==".tsp" #判断是否是文件夹，不是文件夹
         才打开
    df = pd.read_csv(file,sep=" ",hearder=None)
    s.appned(df)
print(s) #打印结果
```

[

A.2 图片

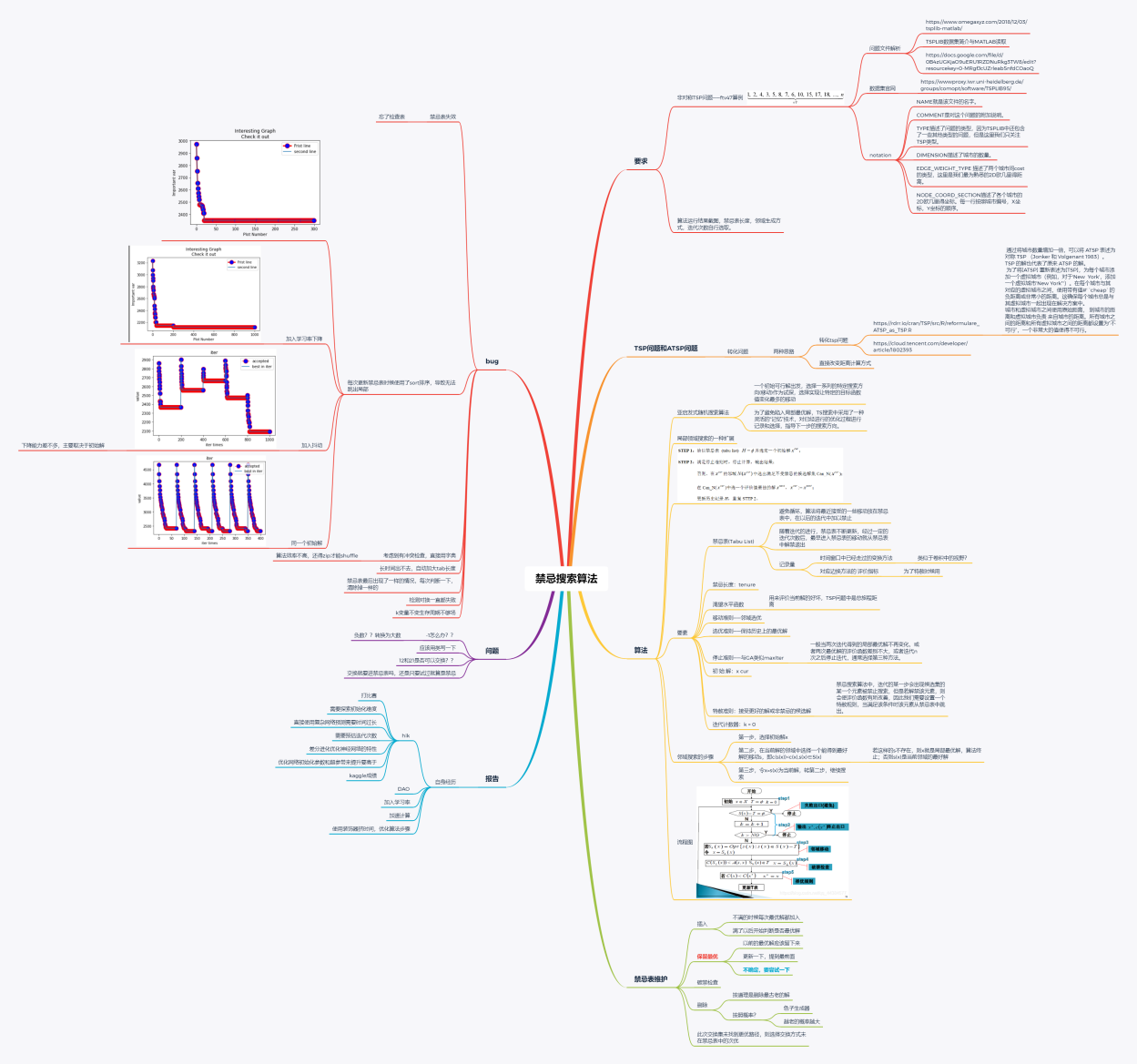


图 13: 禁忌搜索算法求解 TSP 问题思路

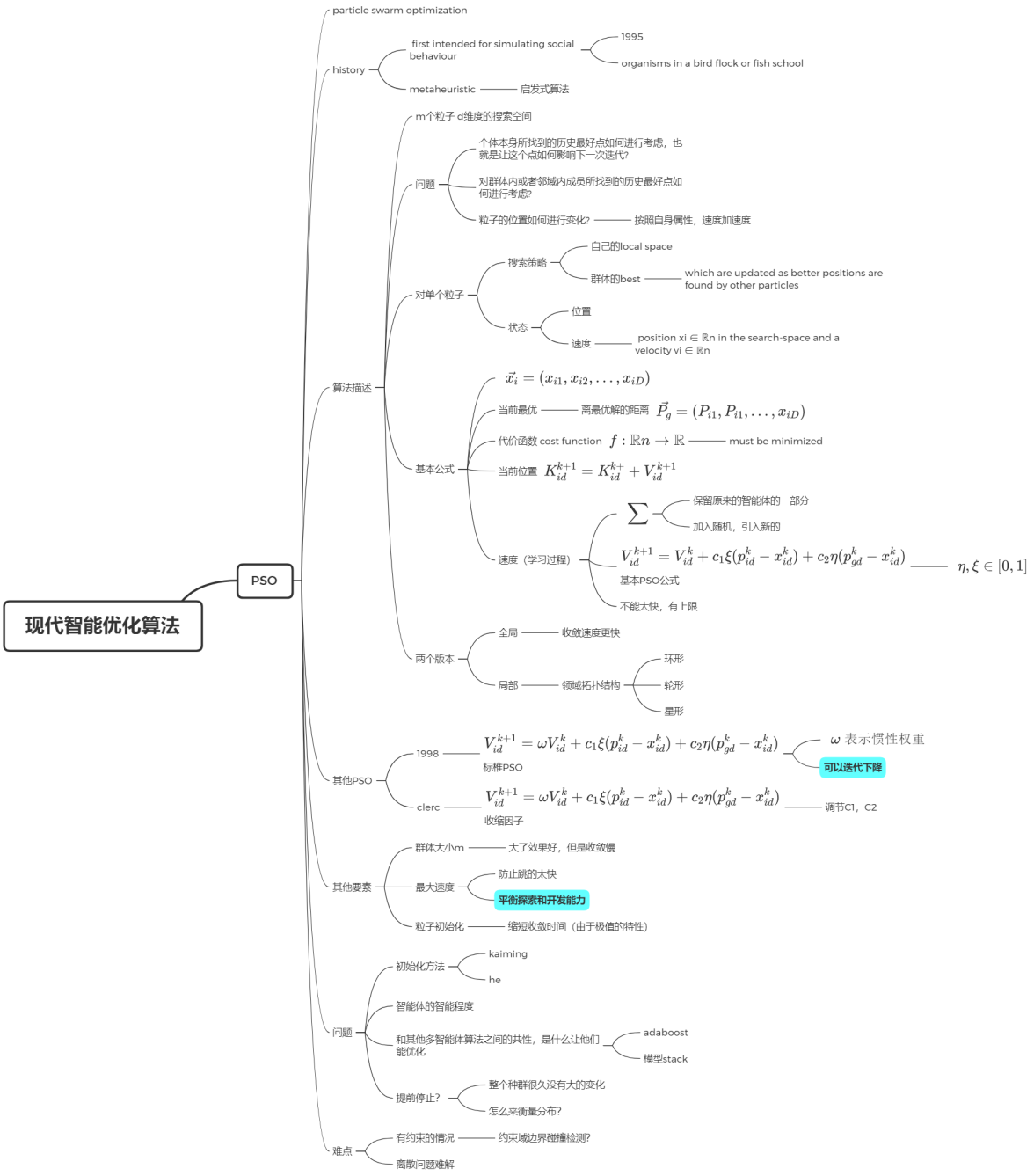


图 14: PSO 求解 TSP 问题思路