“高级算法设计与分析”课程学习报告

21215122, 何峙，大数据与人工智能

本课程把算法原理跟实际问题相结合，将原本枯燥的理论通过一些实际例子阐述明白，十分有趣！不是讲解一个个算法的实现过程，而是深入具体分析一类类算法背后的设计思想，对算法的正确性和合理性进行论证，而且讲解的每类算法之间一环扣一环，既互相区别又互相联系，很具有启发性！现对课程内容进行总结性回顾，以加深对算法分析设计思想的理解。

1. 算法复杂度

这是衡量算法优劣的重要指标。我们不仅要用算法解决问题，而且要越来越快的解决问题，这也是我们学习算法设计最主要驱动力。算法复杂度通常用数量阶表示，如我们熟悉的时间复杂度从小到大的排列依次是：$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^k)$。我们可以跑代码，通过统计监控得出算法的执行时间，但这种统计方式具有很大局限性：1）测试结果受硬件的影响较大；2）测试结果受数据规模的影响很大。本课程介绍了一些快速分析出算法的时间复杂度的方法，如：

（1）递推法：利用递推式不断的循环解嵌套；

（2）换元法：通过代换参数可将复杂的递推式转换为较为简单的递推式；

（3）生成函数法：构造数列的生成函数，再利用数列的性质求解 N 的表达式方法；

（4）特征方程法：通过求解特征方程的根从而求解递推方程的解的方法；

这些方法技巧对快速衡量各种算法的执行效率很有帮助。了解了算法的复杂度，不难认识到随着算法的改进，相同时间内能够处理的数据量也大大提升。

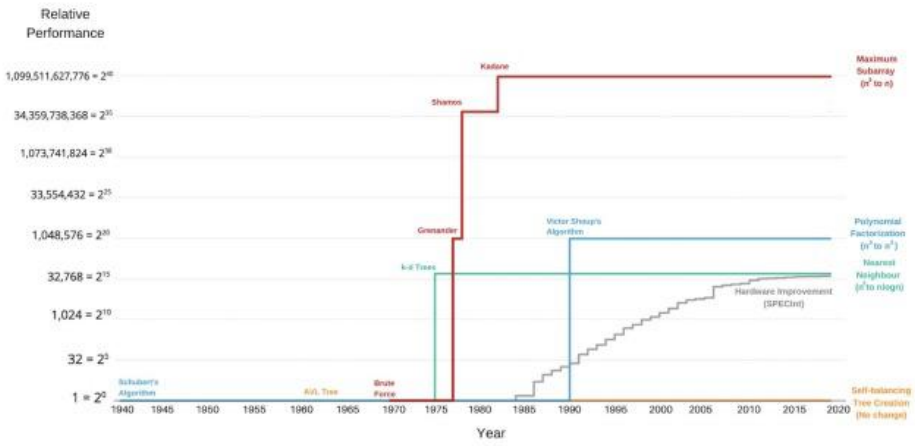

图 1 随年份增长而获得改进的算法对于处理数据量的影响[1]

图 1 列出的 4 个算法族，它们随着年份增长，获得了改进，处理的数据量呈直线式的上升，图中还标有根据摩尔定律的硬件（灰色线）随年份对处理数据量的改进趋势，不难看出对比算法的改进，硬件上的改进对处理数据量的提升显得相当平滑。由此可得：

[1] MIT Computer Science & Artificial Intelligence Laboratory 《How Fast Do Algorithms Improve?》

(1) 算法改进带来的改变问题的可操作性，是硬件改进不能比拟的；

(2) 随着数据量的大量提升，算法的改进比硬件的改进显得更加重要。

2. 分治与递归

分治是一种算法思想，从字面意思即可以理解：当一个问题规模较大且不易求解的时候，就可以考虑将问题分成几个小的模块，逐一解决。而递归是一种算法实践方法，通常跟迭代法相对比：迭代使用的是循环结构，递归则使用选择结构。递归的优点：能使程序的结构更清晰、更简洁、更容易让人理解，从而减少读懂代码的时间。其缺点：但大量的递归调用会建立函数的副本，会消耗大量的时间和内存，而迭代则不需要此种付出。

分治通常跟递归搭配使用会比较让人容易理解，其思维及使用步骤一般为：

(1) 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

(2) 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

(3) 合并：将各个子问题的解合并为原问题的解。

例如本课程作业中的整数划分问题：将正整数划分成一系列小的正整数之和。其递归方程为：

$$q(n, m) = \begin{cases} 1, & n = 1, m = 1 \\ q(n, n), & n < m \\ 1 + q(n, n-1), & n = m \\ q(n, m-1) + q(n-m, m), & n > m > 1 \end{cases}$$

其中 q(n, m)表示最大加数 n 不大于 m 的划分个数，只要我们将正整数划分为小整数的以上 4 种情况都梳理出来，即可以利用递归方法顺利求解。

3. 动态规划

该方法的核心思想是：将待求解的问题分解为若干个子问题，先求解子问题，然后从这些子问题得到原问题的解。经动态规划法得到的子问题往往不是相互独立的，为了避免重复计算，我们可以用一个表来记录所有已解决的子问题的答案。它常用于求解具有最优子结构性质（问题的最优解包含了子问题的最优解）和子问题重叠性质（在用递归算法自顶向下的解决问题时，每次产生的子问题并不总是新问题，有些问题被反复计算多次）的问题。所以，动态规划不同于递归的自顶向下，它是自底向上的，一般有循环迭代完成。

本课程介绍了一些用动态规划解决问题的例子，如构建最优二叉树，矩阵的连乘问题等，其一般解题步骤是：

(1) 定义子问题；

(2) 写出子问题的递推关系；

(3) 依次计算出这些子问题。这里通常会有一个子问题数组，数组的每一个元素对应一个子问题，然后确定这个数组的计算顺序，即可迭代求解；

4. 贪心策略

在对问题求解时，总是采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。它不从整体最优上加以考虑，所做出的是某种意义上的局部最优解。该算法的一般求解步骤可归纳为：

(1) 将问题分解为若干个子问题；

(2) 找出适合的贪心策略；

(3) 求解每一个子问题的最优解，即局部最优解；

(4) 将局部最优解堆叠为全局最优解；

本人认为该算法的唯一难点是找到待求解问题的贪心策略，即贪心策略的选择是否能达到整个问题的最优解。譬如“0-1 背包问题”：有一个背包，背包最大承载总重量是已知且固定，另有有若干个物品，物品有自己的价值和重量，且不可以分割成任意大小。要求尽可能让装入背包中的物品总价值最大，但不能超过背包总重量。对此可以有如下几个贪心策略：

(1) 每次挑选价值最大的物品装入背包？

(2) 每次挑选所占重量最小的物品装入？

(3) 每次选取单位重量价值最大的物品装入？

哪种贪心策略才解除整个问题的最优解，还需证明后才能真正运用到解题中。一般来说，贪心算法的证明围绕着整个问题在贪心策略中存在的子问题的最优解得来的。

或许穷尽所有计算资源也找不到全局最优解，但“轻松”找到局部最优解在现有条件下也未尝不可。

5. 回溯与分枝界限策略

不同于分治、动态规划、贪心策略等式基于规模的算法，回溯与分枝界限是基于搜索的算法。

回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间的树，当搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索（也叫剪枝），逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

分支界限法类似回溯法，也是在问题的解空间上搜索问题解的算法，其求解目标是找出满足约束条件的一个解(而回溯是找出所有的解)，或是在满足条件的解中找出最优解。

这两种算法的容易混淆，主要区别如下：

- 回溯法

(1) 对于求解目标：回溯法的求解目标是找出解空间中满足约束条件的一个解或所有解。

(2) 对于搜索方式：回溯法使用深度优先遍历搜索整个解空间，当不满条件时则丢弃，继续搜索下一个儿子结点，如果所有儿子结点都不满足，向上回溯到它的父节点。

- 分支限界法

(1) 对于求解目标：分支限界法的目标一般是在满足约束条件的解中找出在某种意义下的最优解，也有找出满足约束条件的一个解。

(2) 对于搜索方式：分支限界法以广度优先或以最小损耗优先的方式搜索解空间：

a.广度优先分支界限法：按照队列先进先出原则选取下一个结点为扩展结点；

b.最小损坏分支限界法：按照优先队列规定的优先级选取优先级最高的结点成为当前扩展结点；

6. 局部与随机搜索策略

这类算法是一种启发式的方法，当场景为如下状况时，可考虑使用：

(1) 数据规模大，精确的结果难以在一定时间计算出。

(2) 问题的解可能不精确，但也能够被接受。

(3) 求解的是最优化问题，有一个成本计算模型。

局部搜索算法来源于爬山法，它通过模拟爬山的过程，随机选择一个位置爬山，每次朝着更高的方向移动，直到到达山顶。对照到具体求解问题，每次都在临近的空间中选择最优解作为 当前解，直到局部最优解。这样算法会陷入局部最优解，能否得到全局最优解取决于初始点的位置。初始点若选择在全局最优解附近，则就可能得到全局最优解。算法过程为：从当前的节点开始，和周围的邻居节点的值进行比较。 如果当前节点是最大的，那么返回当前节点，作为最大值(既山峰最高点)；反之就用最高的邻居节点来，替换当前节点，从而实现向山峰的高处攀爬的目的。如此循环直到达到最高点。但局部搜索算法存在几个缺点：

(1) 局部最大：局部最大一般比状态空间中全局最大要小，一旦到达了局部最大，算法就会停止，即便该求解可能并不能让人满意；

(2) 平顶状态：平顶是状态空间中成本函数值基本不变的一个区域，在某一局部点周围成本函数为常量。一旦搜索到达了一个平顶，搜索就无法确定要搜索的最佳方向，会产生随机走动，相当走进了一片大平原，随机走一小步也无法脱离这个地区去到梯度明显的地方，这使得搜索效率降低。

而模拟退火算法来源于固体退火原理，是一种基于概率的算法，将固体加温至充分高，再让其徐徐冷却，加温时，固体内部粒子随温升变为无序状，内能增大，而徐徐冷却时粒子渐趋有序，在每个温度都达到平衡态，最后在常温时达到基态，内能减为最小。它在爬山法的基础上，添加一个概率函数，这个函数能给出一个概率值来决定是否选取该解当前一步骤下的局部最优解，即有一定概率能够跳出局部最优解进而继续寻找全家最优解。由此可见，模拟退火算法的优点在于：不管函数形式多复杂，其有一定概率“跳出”爬山法的“平顶状态”，更有可能找到全局最优解。其算法流程归纳如下：

(1) 初始化：初始温度 T(充分大)，随机初始解状态 S(算法迭代的起点)，每个 T 值的迭代次数 L;

(2) 对 k=1, …, L，迭代如下：

(2.1) 产生新解 S′;

(2.2) 计算增量ΔT=C(S′) - C(S)，其中 C(S)为成本函数;

(2.3) 若ΔT<0 则接受 S′作为新的当前解，否则以概率 exp(-ΔT/T)接受 S′作为新的当前解;

(2.4) 如果满足终止条件(通常取为连续若干个新解都没有被接受时则终止算法)则输出当前解作为最优解，结束程序;

(3) T 逐渐减少，且 T->0，然后转第(2)步；

7. 遗传算法

遗传算法的理论是根据达尔文进化论而设计出来的算法: 人类是朝着好的方向（最优解）进化，进化过程中，会自动选择优良基因，淘汰劣等基因。具体来说，通过复制、交叉、突变等操作产生下一代的解，并逐步淘汰掉适应度函数值低的解，增加适应度函数值高的解。这样进化 N 代后就很有可能会进化出适应度函数值很高的个体。

譬如，使用遗传算法解决“0-1 背包问题”的思路：0-1 背包的解可以编码为一串 0-1 字符串（0：不取，1：取） ；首先，随机产生 M 个 0-1 字符串，然后评价这些 0-1 字符串作为 0-1 背包问题的解的优劣；然后，随机选择一些字符串通过交叉、突变等操作

产生下一代的 M 个字符串，而且较优的解被选中的概率要比较高。这样经过 G 代的进化后就可能会产生出 0-1 背包问题的一个“近似最优解”。其中交叉、变异操作如下：

- 交叉：2 条染色体（0-1 字符串）交换部分基因，来构造下一代的 2 条新的染色体。例如：

交叉前：

00000|011100000000|10000
11100|000001111110|00101

交叉后：

00000|000001111110|10000
11100|011100000000|00101

染色体交叉是以一定的概率（记为 Pc）发生的。

- 变异：在繁殖过程，新产生的染色体中的基因会以一定的概率出错，称为变异。变异发生的概率记为 Pm 。例如：

变异前：

00000111000000010000

变异后：

00000111000010010000

对比与传统优化算法，遗传算法从问题解的集合中开始搜索，而不是单个解开始，从而覆盖面大，利于达到全局最优。而且遗传算法中的选择、交叉和变异都是随机操作，而不是确定的精确规则，这说明遗传算法是采用随机方法进行最优解的搜索，选择体现向最优解的迫近，交叉体现最优解的产生，变异体现全局最优解的覆盖。

通过本课程的学习，更深刻理解到：计算机能做的事只是穷举，而算法则是让计算机可以更有效的去穷举。算法的改进在当今数据分析和机器学习等火热的领域上相当重要，因为它们都依赖于大数据。所以，如何进行算法改进和创新，让算法改进促进计算改进，让更多的指数级别的“慢”算法改进为多项式级别甚至线性级别、常量级别的“快”算法，是一个令人深思的问题。

“旅行商”问题实验报告

21215122，何峙，大数据与人工智能

实验内容

旅行商问题是指一名商人要到若干城市去推销商品，已知城市个数和各城市间的路程（或旅费），要求找到一条从最开始城市出发，经过所有城市且每个城市只能访问一次，最后回到最开始城市的路线，使总的路程（或旅费）最小。

实验设计

本实验使用模拟退火算法求解。模拟退火算法的核心思想为：算法迭代过程中，以一定概率接受比当前解更差的解，然后使用这个更差的解继续搜索。现利用模拟退火算法步骤对旅行商问题建模：

（1）解空间

设经过每个城市形成一个路径为一个解，形成某个排序 s_i = {c1, c2, c3, ...}，其中 ci 代表每个城市，则解空间为所有这些排列的集合 S = {s_1, s_2, s_3, ...}。

（2）初始解

模拟退火算法的最优解与初始解无关，故初始解可为随机生成的一个排列，如 s_0 = {c1, c2, c3, ...}。

（3）目标函数

目标函数即为所有城市的路径总长度（或总旅费）：

$$\text{cost}(c1, c2, ..., cn) = \sum_{i=1}^{n+1} d(ci, c\_i + 1) + d(c1, cn)$$

其中 d 为计算两个城市之间的距离函数。

（4）新解的产生

本实验采用将两个城市的逆序变换作为新解：任选城市序号 j 和 k，交换 j 和 k 的访问顺序，即若交换前解为 s_i = {c1, c2, ..., cj, ..., ck, ..., cn}，则交换后解为 s_i′ = {c1, c2, ..., ck, ..., cj, ..., cn}。

（5）损失函数

目标函数差即为变换前的解和变化后的解的损失函数之差：

$\Delta c = \text{cost}(s\_i') - \text{cost}(s\_i)$

（6）Metropolis 接受准则

接受当前解的概率为 $p = \begin{cases} 1, & \Delta c > 0 \\ exp(-\Delta c / t), & \Delta c < 0 \end{cases}$

其中 t 用于控制退火速度，其迭代规则为 t := a * t，　0< a < 1

实验结果

本实验使用一组随机创建的坐标数据对列表作为每个城市的坐标，形如:

coordinates = [[x1, y1], [x2, y2], ...,]。

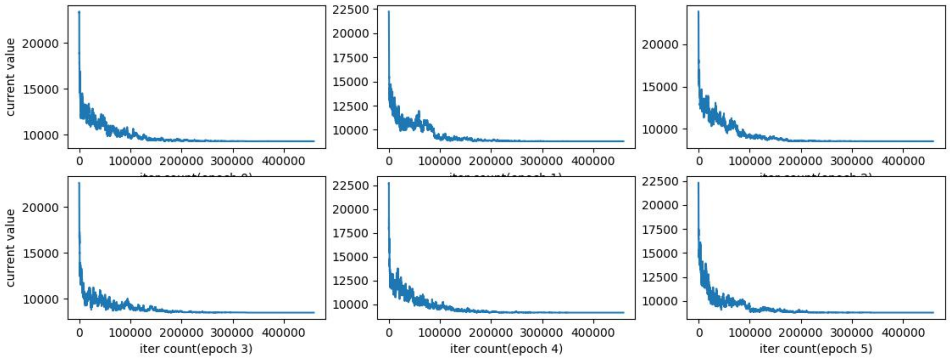

图 1 算法各次求最短距离变化趋势

图 1 显示了跑了 6 次模拟退火算法的旅行最短距离随迭代次数变化的趋势。可以所求最短距离都是不断减少，符合算法预期，即模拟退火算法是可以求解旅行商问题的。

表 1 算法各次求得的最短距离

Epoch	1	2	3	4	5	6
最短距离	9268.03	8816.41	8521.08	8472.09	9136.44	8785.92

表 1 显示每次所求的最短距离都不一样，说明模拟退火算法并不能精确求解，只能迫近最优解。

附代码

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

# 获取点坐标数据
def get_coordinates(file_path: str) -> np.ndarray:
    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]])

    return coordinates
```

```
# 计算点与点之间距离矩阵
def getdismat(coordinates: np.ndarray) -> np.ndarray:
    num = coordinates.shape[0]
    distmat = np.zeros((num, num))
    for i in range(num):
        place_i = coordinates[i]
        for j in range(num):
            place_j = coordinates[j]
            distmat[i][j] = np.sqrt(np.power(place_i[0] - place_j[0], 2) + np.power(place_i[1] - place_j[1],
2))
    return distmat

# 进行模拟退火过程
def sa_run():
    coordinates = get_coordinates('./a280.tsp')
    num = coordinates.shape[0]
    dist_mat = getdismat(coordinates)
    solution_new = np.arange(num)
    solution_current = solution_new.copy()
    solution_best = solution_new.copy()
    value_current = 9999999
    value_best = 9999999
    alpha = 0.99
    t_range = (1, 100)
    markovlen = 1000
    t = t_range[1]

    epochcount = 6
    epoch_best = []
    epoch_current = []
    for epoch in range(epochcount):
        result_best = [] # 记录迭代过程中的最优解
        result_current = []
        while t > t_range[0]:
            for i in range(markovlen):
                # 使用将两个左边逆序的方式产生新解
                while True:
                    loc1 = int(np.ceil(np.random.rand() * (num - 1)))
                    loc2 = int(np.ceil(np.random.rand() * (num - 1)))
                    if loc1 != loc2:
                        break
                solution_new[loc1], solution_new[loc2] = solution_new[loc2], solution_new[loc1]

                value_new = 0
```

```
                for j in range(num - 1):
                    value_new += dist_mat[solution_new[j]][solution_new[j + 1]]
                value_new += dist_mat[solution_new[0]][solution_new[num - 1]]
                if value_new < value_current:
                    # 接受该解
                    # print('accept1')
                    value_current = value_new
                    solution_current = solution_new.copy()

                    if value_new < value_best:
                        value_best = value_new
                        solution_best = solution_new.copy()
                else:
                    # 以一定概率接受该解
                    if np.random.rand() < np.exp(-(value_new - value_current) / t):
                        # print('accept2')
                        value_current = value_new
                        solution_current = solution_new.copy()
                    else:
                        # print('not accept')
                        solution_new = solution_current.copy()

                result_current.append(value_current)

            t = alpha * t
            result_best.append(value_best)
            print(f't: {t}, value: {value_current}')

        epoch_current.append(result_current)
        epoch_best.append(result_best)

        print(f'epoch {epoch} finish!!!!!!!!')
        solution_new = np.arange(num)
        solution_current = solution_new.copy()
        solution_best = solution_new.copy()
        value_current = 9999999
        value_best = 9999999
        t = t_range[1]

    # print(f'best values: {value_best}')
    # print(f'best solution: {solution_best}')

    fig_col = 3  # 每行多少个子图
    fig_row = epochcount // fig_col
```

```
    fig = plt.figure()
    ax = fig.subplots(fig_row, fig_col)
    for r in range(fig_row):
        for c in range(fig_col):
            index = r * fig_col + c
            if c == 0:
                ax[r, c].set_ylabel('current value')
            ax[r, c].set_xlabel(f'iter count(epoch {index})')
            ax[r, c].plot(np.array(epoch_current[index]))

    plt.show()

    for i in range(len(epoch_best)):
        print(f'best value {i}: {epoch_best[i][-1]}')


if __name__ == '__main__':
    sa_run()
```