**算法设计与分析实验报告**

|  |
| --- |
| **实验名称：U204635 推销员问题** |
| **一、问题陈述，相关背景、应用及研究现状的综述分析**  问题重述：在给定图中找到固定起点的线路的最小长度。  研究现状：  旅行商问题，一般称作TSP问题，在规模较大时不可能求得精确解，但U204635就是要求数据规模较小时的精确解。部分资料如下：  旅行商问题(TSP)是一个典型的 NP-hard 组合优化问题，是指给出确定数量的一组城市，要求一个旅行商访问所有的城市一次后返回起始点，目的是以最低的成本(时间、路程等)找到遍历路线，其中所有城市仅被一个旅行商访问一次，起点城市除外[31]。而多旅行商问题(MTSP)则是 m 个旅行商分别访问部分城市，使除起点以外的每个城市仅被一个旅行商访问一次，最终求遍历完所有城市的最低消费(距离、时间等)[12]。当 m=1时，MTSP 就转化为经典 TSP，因此 TSP 是 MTSP 的一种特殊情况。现在像物流“最后一公里”等很多实际问题会涉及到多项任务的分配和优化，所以越来越多的问题被建模为 MTSP，因此 MTSP 受到越来越多的关注和研究[32]。 最初在十九世纪学者们大多采用精确算法对 MTSP 进行求解，比如 Ali 等人用分支定界法对非对称 MTSP 进行求解[33]，求解规模最多为 100 个城市；Gavish 等人对分支定界法进行改进，通过限定分支定界的下界[34]，减少了算法运行所需要的时间。但是问题的解空间随问题规模的扩大呈指数方式增长，而精确算法对此较难求解，因此学者们逐渐倾向于采用启发式算法对 MTSP 进行求解[32]，比如 Wang Y Z 等人提出了一种模因算法对 MTSP 进行求解[35]，并将求解规模由 575 扩大到 1173；Zhou H 等针对多起点闭回路的多旅行商问题，提出了单亲遗传算法并证实了算法求解该问题的优越性[36]；Venkatesh 和 Singh 借鉴生物智能的想法，运用人工蜂群算法和入侵杂草优化算法求解MTSP[37]，提高了求解精度。 求解 MTSP 除了以上算法外，强宁等人还提出了一种加速度粒子群算法[38]，周辉仁等人提出一种改进的差分进化算法[39]，Kencana E N 等人提出了一种蚁群系统来对MTSP 进行求解[40]。随着对 MTSP 研究的不断深入，如何快速高效地对其求解已成为一个研究热点[32]。 |
| **二、模型拟制、算法设计和正确性证明**  状压dp的思想，用一个整数记录已经各个顶点的访问情况，然后每次循环增加路径长度，当路径长度等于n时跳出。具体实现：  def **solve**(n, time\_map):      past = {(0b1, 0): 0}  *# (path, last) -> time*      for \_ in **range**(n - 1):          this = {}          for key, spent in past.**items**():              visited, last = key              for i in **range**(1, n):                  if not visited & 0b1 << i:                      time = time\_map[last][i]                      if time == -1:                          continue                      val = spent + time                      key = visited | 0b1 << i, i                      if key not in this or this[key] > val:                          this[key] = val          past = this      return **min**(past.**values**())  if \_\_name\_\_ == '\_\_main\_\_':      n = **int**(**input**())  **print**(**solve**(n, [[**int**(i) for i in **input**().**split**()] for \_ in **range**(n)]))  这绝对是这道题时间复杂度最低的解法了。 |
| **三、时间和空间复杂性分析**  过程略。显然这比常规的基于dfs的解法的阶乘级时间复杂度小得多。 |
| **四、程序实现和实验测试过程**  **1 验证算法的正确性**  我们搜集了用python语言解决该题并通过洛谷对应测试的四种实现，分别封装成提供数据规模n:int和距离表time\_map:list[list[int]]，返回最短路程长度的函数：  def solve(n:int, time\_map:list[list[int]]) -> int:   ...  我们实现了一个测试点生成器，用于产生不同**数据规模**n、不同**连通率**ratio的邻接矩阵：  from random import randrange ​ def generate\_testcase(n, ratio=1) -> list[list[int]]:    MAX = \_MAX // n    time\_map = [[-1] \* n for \_ in range(n)] ​    while True:        for i in range(n):            time\_map[i][i] = 0 ​        for i, j in combinations(range(n), 2):            time\_map[i][j] = time\_map[j][i] = randrange(MAX) ​        for \_ in range(int(n \* n \* (1 - ratio))):            while True:                i, j = randrange(n), randrange(n)                if i != j and time\_map[i][j] != -1:                    time\_map[i][j] = -1                    break                if check\_connect(time\_map):            return time\_map  其中check\_connect是检测图是否连通的函数，它确保generate\_testcase生成的测试点数据是合理的。  我们在各种大小的地图中，始终确保四个算法的返回值相同。  from solutions.wyh\_solution import solve as solve\_0 from solutions.lx\_solution import solve as solve\_1 from solutions.BF\_solution import solve as solve\_2 from solutions.DP\_solution import solve as solve\_3 ​ args = n, time\_map = ... assert solve\_0(\*args) == solve\_1(\*args) == solve\_2(\*args) == solve\_3(\*args)  在运行了数十亿次不同的testcase中，该断言从未报错，证明四个算法的结果一致，算法正确性得证。  **2 对于给定的输入，比较同类算法的性能**  我们主要关注时间复杂度与数据规模n的关系。  def run(func, \*args, n=10\_000) -> float:    t = time()    for \_ in range(n):        func(\*args)    return time() - t  run函数统计func重复运行同一个测试点n次的耗时。  def do\_test(size, ratio, m=100, n=100):    cases = [        generate\_testcase(size, ratio)        for \_ in range(m)   ] ​    return [        sum(run\_1(func, size, case, n=n) for case in cases) / (m \* n)        for func in (solve\_0, solve\_1, solve\_2, solve\_3)   ]  do\_test函数统计**四个函数**分别跑m个测试点每个n次的**平均单次耗时**。  实际代码还包含一些作图用的脚本，以及一些debug用的助手函数。  四个函数中，BF\_solution.py和lx\_solution.py实现的是**遍历全排列**；wyh\_solution.py是标准的**深度优先搜索**；DP\_solution.py是一种崭新的基于**动态规划**思想的途径。下面先定性分析以上四种算法在不同数据规模n下的平均运行时间。  固定ratio=0.9，令n遍历3到8，比较四种算法耗费的时间，结果作图如下：    固定ratio=1，令n遍历7到13，比较较快两算法耗费的时间，结果作图如下：   1. 在数据量较小时，暴力解法虽然在复杂度上高于深度优先搜索的算法，但是由于其优秀的常数项而得占优势，优势在n<7时得以保持 2. 在数据量较大时，动态规划解法的优势才得以展现。从下图中可以看出，动态规划法的时间复杂度的阶数与深度优先搜索法的阶数有不同     **3 定量分析**  将测试数据序列化保存：  import pickle ​ def run\_and\_save\_result():    x, y\_0, y\_1, y\_2, y\_3 = zip(        \*((size, \*do\_test\_1(size, 1, 20, 20)) for size in range(3, 10))   )    limit = min(map(len, (x, y\_0, y\_1, y\_2, y\_3)))    pickle.dump((       (x[:limit], y\_0[:limit], y\_1[:limit], y\_2[:limit], y\_3[:limit])   ), open("out.pkl", "wb"))  得到out.pkl，包含5行7列的测试结果。以下测试第一步均为读取数据：  import pickle x, y\_0, y\_1, y\_2, y\_3 = pickle.load(open("out.pkl", "rb"))  **3.1 比率测试**    曲线有收敛于常数的趋势，这表明O(n!)的时间复杂度是有合理的。  **3.2 幂测试**    显然，对数变换后仍然不能看成直线，而是下凸的曲线，这表明算法的时间复杂度是超多项式的。 |
| **五、总结**  动态规划法毫无疑问是TSP问题精确解的最快解法。 |