 DOI: https://doi.org/10.12345/bdai.v7i2.37677

Optimizing ACO to solve the traveling salesman problem based on SOA and Floyd algorithm

Hong Gao Yingchun Li

School of Computer and Software Engineering, Liaoning University of Science and Technology, Anshan 114051, Liaoning, China

Abstract

This paper aims at the problem that the ant colony algorithm is prone to fall into local optimality. An improved ant colony algorithm based on the seagull algorithm and the Floyd algorithm is proposed to solve the traveling salesman problem. It fuses the characteristics of the algorithms to construct a new pheromone update mechanism and uses the Floyd algorithm to optimize the initial path. The data experimental results show that this improved algorithm significantly enhances the global search ability and convergence speed, optimizes the solution quality, has obvious advantages in timeliness and accuracy, and has broad application potential.

Keywords

Traveling Salesman Problem; Seagull Optimization Algorithm; Floyd Algorithm; Ant Colony Optimization

基于 SOA 和 Floyd 算法优化 ACO 解决旅行商问题

高宏 李迎春

辽宁科技大学计算机与软件工程学院，中国 · 辽宁 鞍山 114051

摘 要

本文针对蚁群算法容易陷入局部最优等问题，提出一种基于海鸥算法和弗洛伊德算法的改进蚁群算法来求解旅行商问题，融合算法特性构建新信息素更新机制，利用弗洛伊德算法优化初路径。数据实验结果表明，该改进算法显著提升全局搜索能力与收敛速度，优化求解质量，在时效与精度上优势明显，具广泛应用潜力。

关键词

旅行商问题；海鸥算法；弗洛伊德算法；蚁群算法

1 引言

旅行商问题（ Travelling Salesman Problem，TSP ）于物流配送路径规划、智能交通路线优化等领域中，是影响效率与成本的关键因素。随着各行业规模的扩张与精细化管理需求的提升，高效求解 TSP 变得愈发重要。蚁群算法作为求解 TSP 的常用方法，因模拟蚂蚁觅食原理，在路径寻优上有一定优势。然而，在处理大规模 TSP 实例时，蚁群算法常出现收敛速度缓慢、易陷入局部最优解的情况，导致无法在可接受时间内找到全局最优或较优解。

弗洛依德算法以其在求解图中任意两点间最短路径的高效性著称，能够快速构建全面且准确的距离矩阵，为后续路径规划提供坚实基础。海鸥优化算法则是通过模拟海鸥捕食行为，具备强大的全局搜索能力和跳出局部最优的特性。

鉴于此，本文创新性地将弗洛依德算法(Floyd Algorithm)和海鸥优化算法（ Seagull Optimization Algorithm，SOA ）引入蚁群算法中，旨在优化蚁群算法求解 TSP 问题的性能，提升搜索效率与解的质量，为相关领域的实际应用提供更有效的解决方案。

2 算法设计

2.1 蚁群算法的基本原理

蚁群算法（ Ant Colony Optimization, ACO ）是一种模拟蚂蚁觅食行为的群智能优化算法，由 Dorigo 于 1992 年提出 [1]。其灵感源于生物学：蚂蚁在路径上释放“信息素”，后续蚂蚁倾向于选择信息素浓度较高的路径，并通过释放自身信息素进行强化，这种正反馈机制引导蚁群找到最短路径。

【基金项目】本研究得到辽宁科技大学校级大学生创新创业训练计划项目资助。

【作者简介】高宏（2004—），男，中国四川人，在读本科，从事聚焦于启发式智能算法的改进与混合优化策略研究。

在算法模型中，蚂蚁被抽象为智能代理。初始时各路径信息素相等，蚂蚁在构建解时根据状态转移概率选择下一节点，该概率由信息素浓度（历史经验）和启发式信息（如距离倒数，代表局部优劣）共同决定，并通过“赌轮盘”机制平衡开发与探索。一轮搜索结束后，算法执行信息素更新：一方面按速率挥发信息素以避免早熟；另一方面根据解的质量对较优路径进行信息素增强。通过“构建 - 更新”的迭代循环，信息素逐渐集中于优质路径，引导蚁群收敛于最优解。凭借正反馈、并行性和鲁棒性等优势，ACO 被广泛应用于旅行商问题（TSP）、车辆路径问题等组合优化领域。

2.2 海鸥优化算法的基本原理

海鸥算法（Seagull Optimization Algorithm, SOA）是一种群体智能启发式优化算法，由 Gaurav Dhiman 和 Vijay kumar 于 2018 年提出，其核心是模拟海鸥的迁徙、攻击与避碰行为，实现解空间的高效搜索，适用于函数优化、参数辨识等复杂问题 [2]。

算法通过三步行为模型更新候选解：一是迁徙避碰，个体向种群最优位置移动，同时通过线性递减参数调整位置避免碰撞，维持种群多样性；二是螺旋攻击，接近最优区域时，以螺旋运动方程进行局部精细搜索，提升解的精度；三是迭代寻优，通过适应度函数评估位置优劣，循环更新种群位置，最终收敛至全局最优解。

2.3 弗洛伊德算法的基本原理

Floyd 算法（Floyd-Warshall Algorithm）是 1962 年 Robert Floyd 提出的动态规划图论算法 [2]。核心思想是：对于任意的两点 i，j 的最短距离 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]) (k 为中间任意顶点)。

2.4 参数设置

见表 1。

表 1 各城市规模相应参数设置

参数名	30 城市	50 城市	100 城市	150 城市	调整规律说明
ALPHA_SOA	1	1.2	1.5	1.8	随规模增加而增加，增强全局探索能力
BETA_SOA	2	2.5	3	3.5	随规模增加而增加，更重视距离启发式
RHO_SOA	0.4	0.35	0.3	0.25	随规模增加而减少，保留更多探索信息
Q_SOA	80	100	120	150	随规模增加而增加，增强正反馈
ALPHA_ACO	1.2	1.5	1.8	2	随规模增加而增加，强化信息素引导
BETA_ACO	3	4	5	6	显著增加，大规模问题依赖距离信息
RHO_ACO	0.3	0.25	0.2	0.15	显著减少，保留优质路径信息
Q_ACO	120	150	200	250	随规模增加而增加，强化优质解
city_num	30	50	100	150	问题规模
ant_num	40	60	80	100	蚂蚁数量随规模增加，但比例减少

3 ACO&SOA&Floyd 算法

3.1 算法简介

ACO&SOA&Floyd 是融合 Floyd、海鸥优化（SOA）、蚁群（ACO）的混合算法 [4]，核心协作逻辑为：Floyd 算法计算所有城市对最短距离，生成初步的距离矩阵，提高前期收敛速度；SOA 以该矩阵为基准，进一步筛选最有初始矩阵，强化初始信息素分布，加快前期收敛速度，增强全局搜索能力。三者协同提升 TSP 求解效率与精度。

3.2 算法设计

步骤 1：Floyd 算法优化距离矩阵

按欧氏距离公式计算城市间初始距离，填充 distance_matrix；

遍历中间城市 k，执行 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])，得到优化后的最短距离矩阵 Opt_Dist；

识别关键枢纽节点，重点优化枢纽间和枢纽到其他城市的距离；

执行多阶段优化（枢纽间优化 → 枢纽到全图 → 聚类内优化 → 随机采样优化）；

路径重构和平滑处理，得到最终优化距离矩阵。

步骤 2：SOA 优化生成优质初始解

以 Opt_Dist 为依据，将 SOA 解向量转换为 TSP 路径，路径总距离作为适应度值；

采用多策略（最近邻、分层构造、节约算法等）生成多样化初始种群；

迭代执行 SOA 海鸥位置更新（选择、交叉、变异、局部优化），优化得到全局最优路径 SOA_Best_Path 和优质解池 SOA_Solutions；

对 SOA_Best_Path 中相邻城市对，在信息素矩阵对应位置增加强化值，生成 ACO 初始信息素 Init_Pheromone。

步骤 3：ACO 全局搜索

初始化蚁群（蚂蚁数 = 城市数 × 2），随机选择起点，全局最优距离设为无穷大；

以 Init_Pheromone 为初始信息素，Opt_Dist 作为启发式矩阵；

. 以“运行时间未到所设置的秒数”为循环条件，执行：

蚂蚁依据信息素和启发式因子概率选路，遍历所有城市；

计算每只蚂蚁的路径总距离；

更新全局最优解；

按“挥发 + 增量”规则动态更新信息素矩阵；

对全局最优路径执行 2-opt 局部优化，输出最终结果

3.3 参数设置

实验设计与结果分析。

本文提出的 Floyd&SOA&ACO 算法采用 Python 语言 3.11.8 版本在 Pycharm 集成开发环境中实现，为测试该算法求解旅行商问题的性能，并验证其与传统蚁群算法相比的优化效果，实验采用标准城市坐标数据集，选取包含 3 组城市的测试样本，通过计算欧氏距离构建初始距离矩阵，并应用 Floyd 算法进行优化。

为确定适用于旅行商问题的最优算法参数配置，实验进行多轮参数调优，最终确定算法的最优参数如下：海鸥优化算法的种群规模为 30，最大迭代次数为 50，衰减系数为 3.0，信息素增强值为 100；蚁群算法的信息素因子为 1.0、启发函数因子为 2.0、信息素挥发因子为 0.5、信息素释放总量为 100，种群规模为城市数量；

表 2 展示了海鸥优化算法（SOA）、蚁群算法（ACO）、Floyd&ACO、SOA&ACO 与 Floyd&SOA&ACO 算法在城市数为 30、50、100 的数据集上生成的最优路径结果。实验结果表明，在相同参数设置下，Floyd 算法能有效提高 ACO 的收敛速度，SOA 算法能提高收敛速度的同时还能增强全局搜索能力，Floyd&SOA 共同应用于 ACO 算法上，能切实提高求解的效率以及解的质量。

表 2 SOA、ACO、Floyd&ACO、SOA&ACO 算法与 Floyd&SOA&ACO 算法的运行结果

城市数量	运行时间（秒）	SOA		ACO		Floyd&ACO		SOA&ACO		Floyd&SOA&ACO	
		Best	AVG	Best	AVG	Best	AVG	Best	AVG	Best	AVG
30	1	4721	5496	3058	3123	3025	3046	3024	3039	3024	3027
30	5	3520	3709	3024	3025	3024	3024	3024	3024	3024	3024
50	10	6569	6894	3696	3765	3687	3709	3661	3714	3659	3681
50	30	5118	5517	3686	3686	3687	3687	3659	3670	3659	3669
100	5	18716	20026	8782	9034	5213	5321	5218	5286	5145	5297
100	30	13844	14515	5301	5580	5129	5199	5112	5204	5046	5146
100	100	14720	15693	5142	5244	5125	5186	5093	5149	5046	5133

4 结语

针对蚁群算法前期收敛速度较慢且容易陷入局部最优的缺陷，本文引入弗洛伊德算法对路径进行预处理，显著提高了蚁群算法前期的收敛速度；引入海鸥优化算法生成高质量的初始路径并初始化信息素，在加速收敛的同时，有效增强了算法的全局搜索能力，帮助其跳出局部最优。实验数据表明，对于旅行商问题，本文提出的 Floyd&SOA&ACO 算法求解结果优于传统蚁群算法，且命中最优解的概率更高。这证明 Floyd&SOA&ACO 算法融合了弗洛伊德算法的预处理能力、海鸥优化算法的快速收敛能力与海鸥优化算法的全局搜索能力，验证了算法的有效性。

然而，Floyd&SOA&ACO 算法的时间复杂度较高，随着城市规模的增大，其时间复杂度问题越发突出，未来仍需继续优化其时间复杂度。

参考文献

[1] Dorigo M. Ant colony optimization[D]. University of Parma, 1992

[2] Dhiman G, Kumar V. Seagull optimization algorithm: Theory and its applications for large-scale industrial engineering problems[J]. Knowledge-Based Systems, 2019, 165: 169-196.

[3] Floyd R W. Algorithm 97: shortest path[J]. Communications of the ACM, 1962, 5(6): 345.

[4] Wang J, Ersoy O K, He M, et al. Multi-offspring genetic algorithm and its application to the traveling salesman problem[J]. Applied Soft Computing, 2016, 43: 415-423.