2024303053-胡姗-11

算法分析与理论-小组作业-个人工作
2024.11.25

旅行商问题（TSP）

假设有一个旅行商需要访问若干个城市，并且每个城市之间有不同的距离，旅行商需要找到一条最短路径来访问每个城市一次，并最终返回起点。

蚁群优化算法（Ant Colony Optimization）

蚁群优化算法模拟蚂蚁的行为，通过信息素来指引搜索路径，以找到最短路径。
该算法模拟了自然界中蚂蚁觅食的行为，蚂蚁在寻找食物源时，会在其经过的路径上释放一种信息素，并能够感知其他蚂蚁释放的信息素。
信息素浓度的大小表征路径的远近，浓度越高，表示对应的路径距离越短。
正反馈：蚂蚁以较大概率优先选择信息素浓度较高的路径，并释放一定量的信息素，以增强该条路径的信息素浓度。

随机重启爬山算法（Hill Climbing）

爬山算法一种简单的贪心搜索算法，该算法每次从当前解的临近解空间中选择一个最优解作为当前解，直到达到一个局部最优解。
缺点：容易陷入局部最优解
随机重启爬山算法是一种改进的爬山算法，旨在解决爬山算法容易陷入局部最优解的问题。其核心思想是：如果当前的搜索过程没有找到全局最优解，那么就**重新随机生成一个初始状态**（随机重启），继续进行爬山搜索，直到找到全局最优解。

模拟退火算法（Simulated Annealing）

其实也是一种贪心算法，但是它在搜索过程引入了随机因素。
模拟退火算法以一定的概率来接受一个比当前解要差的解，因此有可能会跳出这个局部的最优解，达到全局的最优解。
随机因素：若移动后得到更优解，则总是接收移动；若移动后的解比当前解要差，则以一定的概率接受移动，而且这个概率随着时间推移逐渐降低（逐渐降低才能趋向稳定）
“一定的概率”的计算参考了金属冶炼的退火过程，因此叫做模拟退火算法。

个人负责工作

算法结果可视化模块

```
# _*_ coding : utf-8 _*_
# @Time : 2024/11/6 下午8:03
```

```python
# @Author : Kmoon_Hs
# @File : map_generator

import matplotlib.pyplot as plt
import networkx as nx
from pylab import mpl

# 设置显示中文字体
mpl.rcParams["font.sans-serif"] = ["SimHei"]

# 定义武汉旅游景点及其经纬度
wuhan_scenic_spots = {
    "武汉植物园": (30.537, 114.436),
    "东湖绿道": (30.558, 114.405),
    "欢乐谷": (30.586, 114.425),
    "湖北省博物馆": (30.563, 114.385),
    "武汉大学": (30.536, 114.360),
    "楚河汉街": (30.566, 114.344),
    "黄鹤楼": (30.547, 114.309),
    "武汉长江大桥": (30.562, 114.279),
    "汉口江滩": (30.587, 114.294),
    "归元寺": (30.545, 114.258)
}


# 生成无向带权图，控制每个节点的出度
def generate_weighted_graph_with_edges(spots, problem):
    G = nx.Graph()  # 创建一个无向图
    names = list(spots.keys())

    edges = problem['edges']

    # 将景点作为节点加入图中
    for name in names:
        G.add_node(name, pos=spots[name])

    # 生成边和权重
    for item in edges:
        node1, node2, weight = item
        G.add_edge(names[node1], names[node2], weight=weight)

    return G


def generate_possible_path(spots, result):
    names = list(spots.keys())

    G_directed = nx.DiGraph()
    possible_path = result[0]
    # 定义所有边的默认颜色
```

```
    edge_colors = ["black" for _ in G_directed.edges()]

    # 定义需要更改颜色的边
    special_edges = []
    for i in range(len(possible_path) - 1):
        special_edges.append((names[possible_path[i]], names[possible_path[i +
1]]))

    # 给特定边设置颜色
    for i, edge in enumerate(G_directed.edges()):
        if edge in special_edges or (edge[1], edge[0]) in special_edges:
            edge_colors[i] = "red"  # 将特定边设置为红色
    G_directed.add_edges_from(special_edges)

    return G_directed


# 可视化图
def plot_graph(G, G_directed, spots, problem, algo):
    names = list(spots.keys())
    pos = nx.get_node_attributes(G, 'pos')  # 获取节点的坐标
    weights = nx.get_edge_attributes(G, 'weight')  # 获取边的权重

    # 创建一个多重图来组合无向图和有向图
    G_mixed = nx.MultiDiGraph()
    G_mixed.add_edges_from(G.edges())  # 添加无向边
    G_mixed.add_edges_from(G_directed.edges())  # 添加有向边

    # 定义所有节点的默认颜色
    node_colors = ["skyblue" for _ in G_mixed.nodes()]

    # 定义需要更改颜色的节点
    special_nodes = names[problem['possible_path'][0]]

    # 给特定节点设置颜色
    for i, node in enumerate(G_mixed.nodes()):
        if node in special_nodes:
            node_colors[i] = "orange"  # 将特定节点设置为橙色

    # 绘制无向边
    nx.draw_networkx_edges(G_mixed, pos, edgelist=G.edges(), edge_color="black",
arrows=False)
    # 绘制有向边
    nx.draw_networkx_edges(G_mixed, pos, edgelist=G_directed.edges(),
edge_color="red", arrows=True)
    # 绘制节点
    nx.draw_networkx_nodes(G_mixed, pos, node_color=node_colors,
edgecolors="black")
    nx.draw_networkx_labels(G_mixed, pos)
    # 显示边的权重
```

```
    nx.draw_networkx_edge_labels(G_mixed, pos, edge_labels=weights, font_size=8)

    plt.title(f'Result of {algo}')
    plt.show()


def show(problem, result, algo):
    # 读取 TSP 问题
    print(problem)

    # 生成图
    G = generate_weighted_graph_with_edges(wuhan_scenic_spots, problem)
    G_directed = generate_possible_path(wuhan_scenic_spots, result)

    # 可视化图
    plot_graph(G, G_directed, wuhan_scenic_spots, problem, algo)
```

直方图绘制模块

```
"""
时间分析
"""

import numpy as np
from algo.base import SolutionBase
from utils.plotter import Plotter

from utils import Timer


_plotter = Plotter()
_timer = Timer()


class TimeAnalysis:
    def __init__(self, test_count, algorithm_classes: tuple):
        """
        分析算法平均耗时

        :param test_count: 在**每个**问题 problem 下算法的重复执行次数
        :param algorithm_classes: 算法类元组
        """         self.test_count = test_count
        self.algorithm_classes = algorithm_classes

        # 累积各个算法的耗时
        self.accumulated_times = [0] * len(algorithm_classes)

        # 记录 run 方法被调用了多少次
```

```python
        self.run_count = 0

        # 初始化最终的平均耗时结果
        self.avg_times = [0] * len(algorithm_classes)

        # 算法实例（待初始化）
        self.algorithms: list[SolutionBase] = []

    def _run_once(self):

        for i, algo in enumerate(self.algorithms):
            _timer.start()
            answer = algo.solve()
            _timer.stop()

    def run(self, problem):
        """
        重复运行算法 test_count 次

        :param problem: TSP 问题
        :return: self
        """        # 算法实例初始化
        self.problem = problem
        self.algorithms: list[SolutionBase] = [
            algo_class(problem) for algo_class in self.algorithm_classes
        ]
        self.run_count += 1

        print(f"[Time Analysis] Running algorithms for {self.test_count}
times...")
        for iter in range(self.test_count):
            if (iter + 1) % 5 == 0:
                print(f"Running: {iter + 1}/{self.test_count}", flush=True,
end="\r")
            self._run_once()
        print()
        return self

    def collect(self):
        """
        计算每个算法的平均耗时

        :return: self
        """        for i, time in enumerate(self.accumulated_times):
            # 相当于跑了 test_count*run_count 次
            self.avg_times[i] = time / (self.run_count * self.test_count)
        return self

    def show(self):
        # 算法名
```

```
        algo_names = [algo.algorithm_name for algo in self.algorithms]

        _plotter.bar(
            algo_names,
            self.avg_times,
            title="Average Time of Three Algorithms",
            x_label="Algorithms",
            y_label="Average Time (ms)",
        )

        return self
```

其他工作

- 代码统筹安排、小组分工任务安排
- 汇报 PPT 修改完善
- 课堂分享汇报

FAQ

Q：我看到你们的输入用例中的图是从稀疏到稠密的，请问这个稀疏和稠密是怎么定义的?
A：最稀疏的图即刚好连通的一个单环，最稠密的图是完全图。前者有 n 条边，后者有 n(n-1)/2 条边，比如稠密度 25% 指的是图中有 n+(n(n-1)/2-n) * 0.25 条边。

Q：按照你的定义，非 100% 稠密的图不是完全图，算法中对于非完全图是怎么去处理那些"不存在"的边的?
A：对于不存在的边，我们将这些边的边权设置为了一个**很大的值**，这样一来算法在迭代过程中几乎就不会去选择这些边。实验证明我们这样做确实是对的。