Lab2-最短路的 Dijkstra 算法

吴欣怡 PB21051111

2024 年 1 月 11 日

问题介绍

在图中, 不可避免要解决的一个问题就是计算两点之间的最短路径, 对于图结构来说，两个点之间不一定只有一条路径，如何能找出最短的那一条路径就是图结构中的最短路径问题。最短路径问题在实际生活中应用十分广泛。这次实验实现了用 dijkstra 算法和线性规划求解器求解最短路问题。

实验要求如下:

1. 实验要求实现最短路的 Dijkstra 算法
2. 能够判断图是否连通，是否有负权边
3. 测试算法时间与图中节点的数量关系，和线性规划求解时间做对比

算法原理

Dijkstra 算法

Dijkstra 算法是基于贪心思想解决单源最短路径问题的方法。对于有向图 G，其中包含节点集合 V 和边集合 E，每条边 (u, v) 都有一个非负权重 $w(u, v)$。定义:

- s 是源节点

- $d[v]$ 表示从源节点 s 到节点 v 的最短路径长度

算法核心思想：

1. 初始化距离数组 $d[]$，将 s 到 s 的距离设为 0，其他节点的距离设为无穷大。
2. 重复以下步骤直到所有节点都被访问：
    - 从未访问的节点中选择距离最小的节点 u。
    - 对于 u 的每个邻居节点 v，更新 $d[v] = \min(d[v], d[u] + w(u, v))$。

线性规划求解最短路径

考虑有向图 G，其中包含节点集合 V 和边集合 E，每条边 (u, v) 都有一个权重 c_{uv}。

定义：

- x_{uv} 是决策变量，表示是否选择从节点 u 到节点 v 的边。
- c_{uv} 是目标函数系数，表示从节点 u 到节点 v 的边的权重。

线性规划模型：

$$\text{Minimize} \sum_{(u,v)\in E} c_{uv} \cdot x_{uv}$$

约束条件：

1. 每个节点的流量守恒：
$$\sum_{(u,v)\in E} x_{uv} - \sum_{(v,u)\in E} x_{vu} = \begin{cases} 1, & \text{如果}u\text{是源点} \\ -1, & \text{如果}u\text{是汇点} \\ 0, & \text{otherwise} \end{cases}$$
2. 边的取值范围：
$$0 \leq x_{uv} \leq 1, \forall (u, v) \in E$$

目标函数表示要最小化所选边的总权重，约束条件确保了流量守恒和边的取值范围。通过求解这个线性规划模型，可以得到图中的最短路径。

编译环境及使用方法

使用 python3.11 环境，采用 python 的 networkx 以点数 n 和两点连接的概率 p 为参数，随机生成一个图。dijkstra 算法中使用堆（heapq）来有效地选择最小距离的节点，并更新与该节点相邻的节点的距离。调用 coptpy 辅助实现线性规划求解最短路，进行两种算法的时间对比。

dijkstra 算法

1. dijkstra(graph, start) 使用堆（heapq）来有效地选择最小距离的节点，并更新与该节点相邻的节点的距离。
2. generate_er_graph(n, p) 以点数 n 和两点连接的概率 p 为参数，随机生成一个图
3. is_connected(graph) 检查图是否连通
4. check_all_edges(graph) 检查有无负权边

调用 coptpy 的线性规划实现最短路

使用 COPT 库创建线性规划模型，其中定义了变量、目标函数和约束条件，然后通过求解器进行求解。

调用 dijkstra 和求解器

对指定规格的图调用 dijkstra 和线性规划的相关函数，对 20 个有效（无负权且连通）的图运行，取时间的平均。

数据集说明

无原始数据集，实验使用的图是调用 networkx 包生成包含随机边的连通图。

测试结果

1. 检验连通图及负权边

能够检验出是否连通（如下图）：

```
graph=generate_er_graph(100, 0.01)
print(is_connected(graph))
```

```
False
```

```
graph=generate_er_graph(100, 0.3)
print(is_connected(graph))
```

```
True
```

能够检验出是否有负权边（如下图是一个有负权边的情况）。在实际代码中，为了方便，把边权设置为从正数中随机选择。

```
In [11]: graph=generate_er_graph(100, 0.3)
         for edge in graph.edges:
                 print(graph.edges[edge]['weight'])
         check_all_edges(graph)
```

```
-5
0
4
7
2
1
1
0
-2
8
-9
-9
-1
-9
6
7
9
```

```
Out[11]: True
```

2.dijkstra 和线性规划对比

以下分别为 dijkstra 和线性规划求解器在规模为 100、2000、10000 个点的连通图上求解最短路的平均耗时。

100 个点：

```
Dijkstra求出规模为100个点的图的最短路径耗时:
0.0004743218421936035
线性规划求解规模为100个点的图的最短路径耗时:
0.015885090827941893
```

2000 个点：

```
Dijkstra求出规模为2000个点的图的最短路径耗时:
0.013497078418731689
线性规划求解规模为2000个点的图的最短路径耗时:
0.026455998420715332
```

10000 个点：

```
Dijkstra求出规模为10000个点的图的最短路径耗时:
0.12821949720382692
线性规划求解规模为10000个点的图的最短路径耗时:
0.10888655185699463
```

分析与总结

可以看出 dijkstra 算法在点的数量较小的时候求解效率显著优于用求解器求解。但是在结点数较多、图的规模比较大的时候求解时间更接近。

A Computer Code

```
import heapq
import networkx as nx
import random
import numpy as np
import time
from coptpy import *
# 参数
n = 100
epsilon = 0.2

def create_spp_model(graph, source):
    env = Envr()
    model = env.createModel("SPP model")
    x = {}
    for edge in graph.edges:
```

```
        x[edge] = model.addVar(vtype=COPT.BINARY, name=f'x_{edge
            [0]}_{edge[1]}')
    obj = LinExpr()
    for edge in graph.edges:
        obj.addTerms(x[edge], graph.edges[edge]['weight'])
    model.setObjective(obj, COPT.MINIMIZE)
    for node in graph.nodes:
        if node != source:
            lhs = LinExpr()
            for edge in graph.edges:
                if node in edge:
                    coeff = 1 if edge[1] == node else -1
                    lhs.addTerms(x[edge], coeff)
            model.addConstr(lhs == 0, name=f'flow_conservation_{
                node}')
    lhs_source = LinExpr()
    for edge in graph.edges:
        if source in edge:
            lhs_source.addTerms(x[edge], 1)
    model.addConstr(lhs_source == 1, name='
        flow_starts_from_source')

    return model

def dijkstra(graph, start):
    heap = [(0, start)]
    visited = set()
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0

    while heap:
        current_distance, current_vertex = heapq.heappop(heap)

        if current_vertex in visited:
            continue

        visited.add(current_vertex)

```

```python
        for neighbor, edge_data in graph[current_vertex].items():
            weight = edge_data['weight']
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances

def is_connected(graph):
    # 连通性检验
    visited = set()
    stack = [random.choice(list(graph.nodes))]

    while stack:
        current_vertex = stack.pop()
        visited.add(current_vertex)

        for neighbor in graph.neighbors(current_vertex):
            if neighbor not in visited:
                stack.append(neighbor)

    return len(visited) == len(graph.nodes)

def check_all_edges(graph):
    # 检查有无负权边
    for edge in graph.edges:
        if graph.edges[edge]['weight'] < 0:
            return True
    return False

def generate_er_graph(n, p):
    #生成图
    graph = nx.erdos_renyi_graph(n, p)
    for edge in graph.edges:
        graph.edges[edge]['weight'] = random.randint(1, 10)
```

```python
    return graph

# 运行Dijkstra算法和线性规划算法
def run(n):
    k = 20
    dijkstra_time_set = 0.0
    lp_time_set = 0.0

    while k >= 0:
        p_connected = ((1 + epsilon) * np.log(n)) / n + 0.001
        graph_connected = generate_er_graph(n, p_connected)
        if is_connected(graph_connected):
            k -= is_connected(graph_connected)
        else:
            graph_connected = generate_er_graph(n, p_connected)
        if check_all_edges(graph_connected):
            continue
        start_node = random.choice(list(graph_connected.nodes))
        source_node = random.choice(list(graph_connected.nodes))

        # Dijkstra算法
        starttime = time.time()
        dijkstra_distances_connected = dijkstra(graph_connected,
            start_node)
        endtime = time.time()
        dijkstra_time_set += endtime - starttime

        # 线性规划求解
        spp_model = create_spp_model(graph_connected, start_node
            )
        start_time = time.time()
        spp_model.solve()
        end_time = time.time()
        lp_time_set += end_time - start_time

    print("Dijkstra求出规模为%s个点的图的最短路径耗时:" % n)
    print(dijkstra_time_set / 20)
    print("线性规划求解规模为%s个点的图的最短路径耗时:" % n)
```

```
    print(lp_time_set / 20)
#三种不同规模的图
run(100)
run(2000)
run(10000)
```