运筹学报告——Dijkstra 求单源最短路径

申长硕 @PB22020518
AI&DS
stephen_shen@mail.ustc.edu.cn

2024 年 12 月 29 日

摘要

本实验旨在通过实现 Dijkstra 算法解决最短路径问题，重点探讨其在连通图上的性能表现，并与线性规划（LP）建模方法进行对比分析。实验中，我们首先利用 NetworkX 库生成 Erdős–Rényi（ER）随机图，确保图的连通性并随机赋予边权重。随后，我们设计并实现基于 'heapq'的 Dijkstra 算法，避免直接使用 'PriorityQueue'。此外，通过调用外部求解器（如 COPT）对 LP 建模方法进行求解，并与 Dijkstra 算法在不同规模图上的运行时间进行对比。实验结果将展示两种方法在求解效率上的差异，并分析各自的适用场景和优缺点。

1 背景与目标

最短路径问题是图论研究中的经典问题，在网络路由、交通规划、物流调度等领域具有广泛的应用。Dijkstra 算法作为最优子结构和贪心策略的典型代表，长期以来被认为是解决单源最短路径问题的高效方法。另一方面，线性规划（LP）建模方法通过全局优化的手段，也能够精确求解此类问题。两种方法在理论基础和实现方式上存在显著差异，其运行效率和适用场景也各有特点。

本实验的主要目标包括以下几点：

1. 实现基于 'heapq'的 Dijkstra 算法，并验证其在随机生成的连通图上的性能。
2. 利用 NetworkX 生成 ER 随机图，确保图连通并随机赋予边权重，分析算法在不同图规模下的表现。
3. 使用线性规划建模求解最短路径问题，通过求解器（如 COPT）获取最优解，并与 Dijkstra 算法进行效率对比。
4. 总结两种方法的优缺点，探讨其在实际场景中的适用性。

通过本实验，我们希望能深入理解最短路径问题的不同求解方法及其性能差异，为算法设计与优化提供理论和实践支持。

2 方法与实现

2.1 实验设计

- 图的生成：使用 `ERGraph` 模块生成随机连通的 Erdős-Rényi 图（ER 图），节点数为 100，边的连接概率为 0.1，边权重为随机值，范围为 $[0, 10)$。
- 算法实现：
    1. Dijkstra 算法采用最小堆 (`heapq`) 实现，通过维护未访问节点的最短距离，逐步找到从起点到终点的最短路径。

2. 线性规划方法将最短路径问题建模为线性规划，通过 `scipy.optimize.linprog` 求解，解析决策变量获得路径。

- 运行效率：记录两种算法的运行时间，并分析其效率差异。
- 结果一致性：验证两种算法的路径长度及具体路径是否一致。

2.2 算法实现

2.2.1 Dijkstra 算法

以下是 Dijkstra 算法的核心实现，基于最小堆动态更新节点的最短距离：

Listing 1: Dijkstra 算法实现

```python
def dijkstra(graph, start_node, end_node=None):
    num_nodes = len(graph)
    distances = [float('inf')] * num_nodes
    predecessor = [-1] * num_nodes
    distances[start_node] = 0
    visited = [False] * num_nodes
    heap = []
    heapq.heappush(heap, (0, start_node))  # (距离, 节点)

    while heap:
        current_distance, current_node = heapq.heappop(heap)
        if visited[current_node]:
            continue
        visited[current_node] = True
        if end_node is not None and current_node == end_node:
            break
        for neighbor in range(num_nodes):
            weight = graph[current_node][neighbor]
            if weight > 0:
                distance = current_distance + weight
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    predecessor[neighbor] = current_node
                    heapq.heappush(heap, (distance, neighbor))
    return distances, predecessor
```

2.2.2 线性规划方法

以下是线性规划方法的核心实现，通过构造约束矩阵将最短路径问题建模为线性规划：

Listing 2: 线性规划方法实现

```python
def solve_lp(graph, start_node, end_node):
    num_nodes = len(graph)
    num_edges = np.sum(graph > 0)
    edges = np.argwhere(graph > 0)
    costs = graph[graph > 0]

    A_eq = np.zeros((num_nodes, num_edges))
```

```
    b_eq = np.zeros(num_nodes)
    for i, (u, v) in enumerate(edges):
        A_eq[u, i] = 1
        A_eq[v, i] = -1
    b_eq[start_node] = 1
    b_eq[end_node] = -1
    bounds = [(0, 1) for _ in range(num_edges)]

    result = linprog(costs, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')
    if result.success:
        x = result.x
        path = [start_node]
        current_node = start_node
        while current_node != end_node:
            for i, (u, v) in enumerate(edges):
                if u == current_node and x[i] > 0.5:
                    path.append(v)
                    current_node = v
                    break
        return result.fun, path
    else:
        raise ValueError("Linear programming failed.")
```

3 结果与分析

3.1 实验结果

以下是实验的部分结果，比较了 Dijkstra 算法与线性规划方法的路径长度、路径及运行时间：

编号	连通度	Dijkstra 距离	Dijkstra 路径	Dijkstra 用时 (s)	LP 路径长度	LP 用时 (s)
1	1.716	4.957	[0, 98, 92, 99]	0.000653	4.957	0.010026
2	1.770	2.503	[0, 72, 26, 99]	0.000119	2.503	0.005075
3	2.550	6.487	[0, 9, 40, 85, 99]	0.000474	6.487	0.005972
4	0.888	5.467	[0, 70, 88, 99]	0.000695	5.467	0.005017
5	2.388	7.496	[0, 94, 71, 68, 69, 54, 74, 95, 99]	0.001125	7.496	0.006750

表 1: 实验结果对比

3.2 结果分析

- ** 路径长度与路径一致性 **：实验结果表明，在随机生成的有限的 case 下，Dijkstra 算法与线性规划方法的路径长度和具体路径完全一致，验证了两种算法的正确性。
- ** 效率对比 **：
  - Dijkstra 算法运行时间显著快于线性规划方法，对 100 节点的图，Dijkstra 运行时间通常在 1 毫秒以内，线性规划运行时间在 5 – 10 毫秒之间。
  - 线性规划方法效率较低的原因在于需要构造约束矩阵并调用通用求解器，复杂度较高。

3.3 算法效率的可视化

以下图展示了 Dijkstra 算法与线性规划方法的运行时间对比：

```
(base) shenc@tk:~/Desktop/Study/OR-lab/lab2/src$ python dijkstra_lp.py
The connected Erdos-Renyi graph is generated. Algebraic Connectivity: 1.770, size of 100
图已验证为连通。
Dijkstra算法：最短路径长度 = 2.5028623436086272, 用时 = 0.000119 秒
Dijkstra算法：从节点 0 到节点 99 的路径 = [0, 72, 26, 99]
线性规划：最短路径长度 = 2.5028623436086272, 用时 = 0.005075 秒
线性规划：从节点 0 到节点 99 的路径 = [0, 72, 26, 99]
效率比较：Dijkstra用时 = 0.000119 秒，LP用时 = 0.005075 秒
(base) shenc@tk:~/Desktop/Study/OR-lab/lab2/src$ python dijkstra_lp.py
The connected Erdos-Renyi graph is generated. Algebraic Connectivity: 2.550, size of 100
图已验证为连通。
Dijkstra算法：最短路径长度 = 6.4872426761191315, 用时 = 0.000474 秒
Dijkstra算法：从节点 0 到节点 99 的路径 = [0, 9, 40, 85, 99]
线性规划：最短路径长度 = 6.4872426761191315, 用时 = 0.005972 秒
线性规划：从节点 0 到节点 99 的路径 = [0, 9, 40, 85, 99]
效率比较：Dijkstra用时 = 0.000474 秒，LP用时 = 0.005972 秒
(base) shenc@tk:~/Desktop/Study/OR-lab/lab2/src$ python dijkstra_lp.py
The connected Erdos-Renyi graph is generated. Algebraic Connectivity: 0.888, size of 100
图已验证为连通。
Dijkstra算法：最短路径长度 = 5.467091670328982, 用时 = 0.000695 秒
Dijkstra算法：从节点 0 到节点 99 的路径 = [0, 70, 88, 99]
线性规划：最短路径长度 = 5.467091670328982, 用时 = 0.005017 秒
线性规划：从节点 0 到节点 99 的路径 = [0, 70, 88, 99]
效率比较：Dijkstra用时 = 0.000695 秒，LP用时 = 0.005017 秒
(base) shenc@tk:~/Desktop/Study/OR-lab/lab2/src$ python dijkstra_lp.py
The connected Erdos-Renyi graph is generated. Algebraic Connectivity: 2.388, size of 100
图已验证为连通。
Dijkstra算法：最短路径长度 = 7.495906129197551, 用时 = 0.001125 秒
Dijkstra算法：从节点 0 到节点 99 的路径 = [0, 94, 71, 68, 69, 54, 74, 95, 99]
线性规划：最短路径长度 = 7.495906129197551, 用时 = 0.006750 秒
线性规划：从节点 0 到节点 99 的路径 = [0, 94, 71, 68, 69, 54, 74, 95, 99]
效率比较：Dijkstra用时 = 0.001125 秒，LP用时 = 0.006750 秒
```

图 1: Dijkstra 算法与线性规划方法运行时间对比

4 总结与反馈

4.1 总结

- 实验通过 Dijkstra 算法与线性规划方法求解最短路径问题，验证了两种方法的正确性和一致性。
- Dijkstra 算法对稀疏图的运行效率显著优于线性规划方法，适合大规模图的最短路径求解。
- 线性规划方法尽管效率较低，但适合更复杂约束的路径问题，具有更好的通用性。

4.2 改进

- Dijkstra 算法：
  - 可扩展为支持负权边的 Bellman-Ford 算法。
  - 在稠密图中，可结合 A* 算法增加启发式优化。
- 线性规划方法：
  - 使用更高效的求解器（如 Gurobi 或 COPT）提升性能。
  - 优化约束矩阵的构造，减少求解器的输入规模。