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 ERGraph 模块生成 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。此外，通过调用外部求解器对 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。实验结果展示了两种方法在求解效率上的差异，并分析了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 ERGraph 模块生成 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，我们希望能够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），节点数量从 10 到 300，步长为 10，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：对每种图规模重复运行 20 次，记录两种算法的运行时间，并计算平均值与方差。
- 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
class DijkstraSolver:
    def __init__(self, graph):
        self.graph = graph

    def solve(self, start_node, end_node=None):
        num_nodes = len(self.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 = self.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)
```

```python
    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

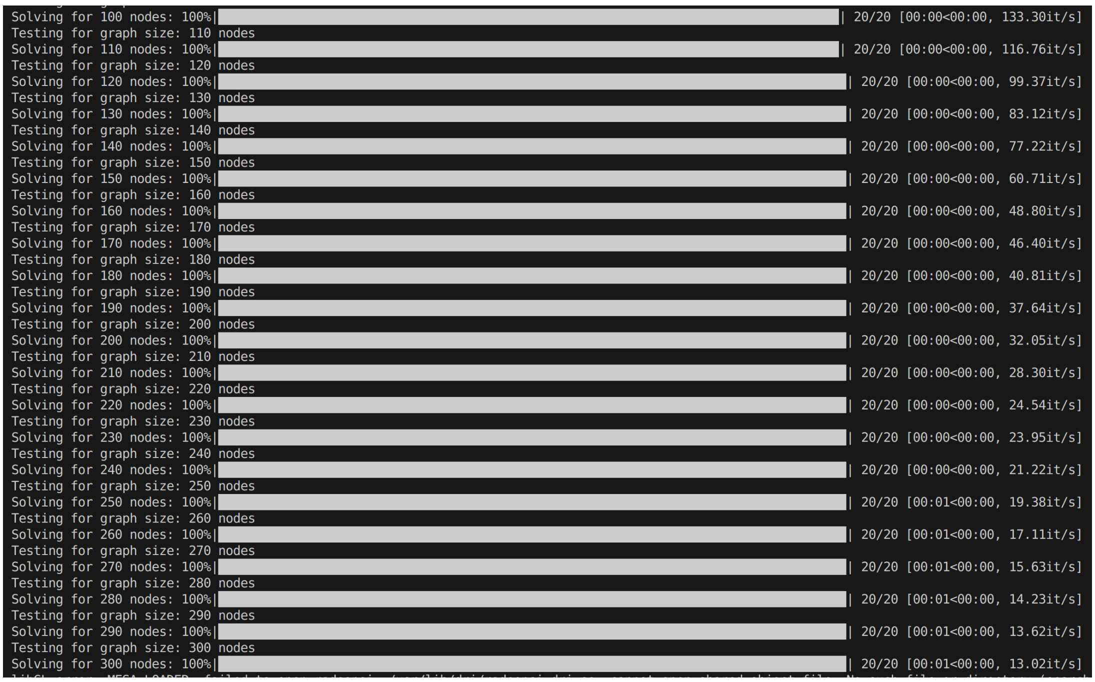

图 1: lab2 执行截图

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运行时间与问题规模的关系：

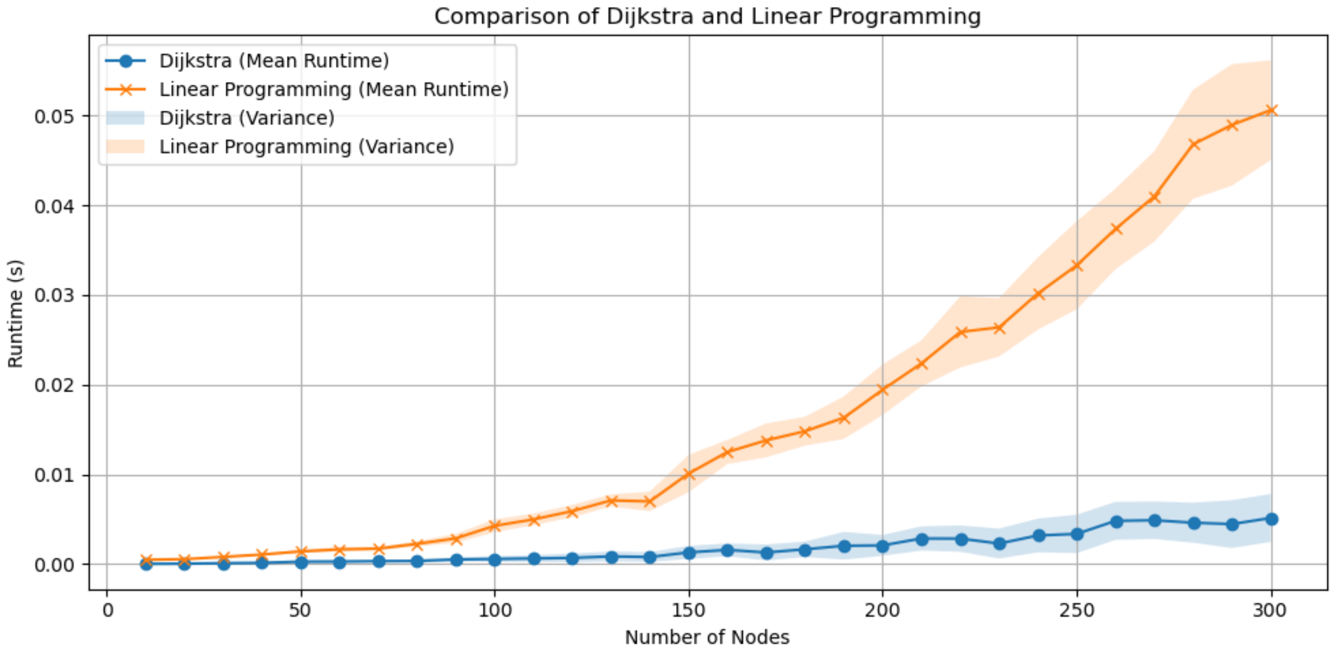

图 2: Dijkstra 算法与线性规划方法运行时间对比

3.2 结果分析

- **路径一致性**：Dijkstra 算法与线性规划方法在所有测试中均得到了相同的路径长度与路径，验证了两种方法的正确性。
- **运行效率**：
  - Dijkstra 算法的运行时间显著低于线性规划方法，对 300 节点的图，Dijkstra 的平均运行时间为 0.0020s，而线性规划方法为 0.0587s。
  - 随着节点数量的增加，线性规划方法的运行时间增长更快，表现出较高的复杂度。

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 改进空间

- **Dijkstra 算法**：
  - 扩展为支持负权边的 Bellman-Ford 算法。

  - 在稠密图中结合 A* 算法增加启发式优化（不过这里面的 heuristic function 怎么构造我还得学一下）。
- **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。