# 王译霆

Dijkstra 算法是一种用于查找单源最短路径的经典算法，假设图中所有边的权重都是非负的。标准的 Dijkstra 算法通过每次选择当前未处理节点中距离源点最近的节点来确保一旦一个节点被“访问”后，其最短路径已经确定，因此通常每个节点只会被处理一次。

然而，在某些应用场景中，可能需要允许算法在构建路径时多次经过同一个节点。例如，可能存在需要绕行特定节点或在特定条件下允许重复经过节点的需求。要改良 Dijkstra 算法以允许重复经过某个节点，可以考虑以下几种方法：

## 1. 允许节点多次入队

标准 Dijkstra 算法在处理每个节点时，会将其标记为“已访问”，并且不再对其进行更新。要允许节点被多次访问，可以移除“已访问”标记。这意味着当找到一条更短的路径到某个已经处理过的节点时，该节点仍然会被重新加入优先队列，以探索通过该节点延伸出的其他路径。

**步骤如下：**

1. **初始化**：与标准 Dijkstra 算法相同，初始化所有节点的距离为无穷大，源点的距离为0。
2. **优先队列**：使用一个优先队列（通常是最小堆）来选择当前距离最短的节点。
3. **节点处理**：
   - 从优先队列中提取距离最短的节点 `u`。
   - 遍历 `u` 的所有邻居 `v`，计算从源点经过 `u` 到 `v` 的新距离。
   - 如果新距离小于已记录的 `v` 的距离，更新 `v` 的距离并将 `v` 重新加入优先队列。
4. **重复**：由于不再标记节点为“已访问”，相同的节点可以被多次加入和处理，允许路径中重复经过该节点。

**示例代码（伪代码）**：

```
function modifiedDijkstra(graph, source):
    distance = [∞] * graph.number_of_nodes
    distance[source] = 0
    priority_queue = MinHeap()
    priority_queue.insert(source, 0)

    while not priority_queue.is_empty():
        u, dist_u = priority_queue.extract_min()

        for each neighbor v of u:
            alt = dist_u + weight(u, v)
            if alt < distance[v]:
                distance[v] = alt
                priority_queue.insert(v, alt)

    return distance
```

## 2. 限制重复次数

为了防止算法因为允许无限次重复访问节点而陷入无限循环，可以引入一个限制，限定每个节点在路径中被访问的最大次数。例如，对于一个图中的每个节点，可以记录它被访问的次数，并在超过限制时跳过。

**步骤如下：**

1. **状态扩展**：每个节点的状态不仅包括其名称，还包括它在当前路径中被访问的次数。
2. **优先队列**：在优先队列中存储节点的状态，包括节点和访问次数。
3. **访问限制**：在处理节点时检查其被访问的次数，超过限制则跳过。

**示例代码（伪代码）**：

```
function modifiedDijkstraWithLimit(graph, source, max_visits):
    distance = [ [∞] * (max_visits + 1) for _ in graph.nodes ]
    distance[source][1] = 0
    priority_queue = MinHeap()
    priority_queue.insert((source, 1), 0)

    while not priority_queue.is_empty():
        (u, visits_u), dist_u = priority_queue.extract_min()

        for each neighbor v of u:
            new_visits = visits_u + 1
            if new_visits > max_visits:
                continue
            alt = dist_u + weight(u, v)
            if alt < distance[v][new_visits]:
                distance[v][new_visits] = alt
                priority_queue.insert((v, new_visits), alt)

    return min(distance[target][1:max_visits+1])
```

## 3. 使用其他算法

如果允许重复经过节点，且路径可能涉及多次绕行，可能需要考虑使用其他算法，如 **广度优先搜索（BFS）**（适用于无权图）或 **A* 搜索算法**，结合合适的启发式函数，以找到更优化的路径。

## 注意事项

- **无负权边**：确保图中所有边的权重依然是非负的，否则算法可能无法正确终止或找到最短路径。
- **循环检测**：允许重复访问节点可能会引入循环，特别是在有相同或更短路径存在时。通过限制重复次数或其他方法来避免算法陷入无限循环。
- **性能考虑**：允许节点多次访问会增加算法的时间复杂度和空间复杂度，特别是在大型图中。因此，应根据具体需求权衡算法的可行性。

## 总结

通过移除或修改标准 Dijkstra 算法中的“已访问”机制，允许节点多次入队和处理，可以实现允许重复经过节点的需求。然而，需要谨慎处理可能引入的循环和性能问题，根据具体应用场景选择合适的改良方法。