南開大學
Nankai University

计算机学院

算法设计大作业实验报告

k 短路问题

姓名：杨涵

学号：2213739

专业：计算机科学与技术

2024 年 6 月 7 日

目录

1 问题描述

给定一个有 n 个结点，m 条边的有向图，求从 s 到 t 的所有不同路径中的第 k 短路径的长度。

前置知识

- 图 (graph) 是一个二元组 $G = (V(G), E(G))$。其中 $V(G)$ 是非空集，称为 **点集 (vertex set)**，对于 V 中的每个元素，我们称其为 **顶点 (vertex)** 或 **节点 (node)**，简称点；$E(G)$ 为 $V(G)$ 各结点之间边的集合，称为 **边集 (edge set)**。
- 常用 $G = (V, E)$ 表示图。
- 对于带边权的图，任意一条路径的边权之和称为这条路径的长度。
- 最短路径问题是指求任意一对节点间所有路径中最小的一条，常见的单源最短路算法有 Bellman—Ford、Dijkstra、Spfa，常见的多源最短路径算法为 floyd 算法。

本次算法研究中会基于 Dijkstra 进行算法扩展，由于 Dijkstra 算法无法处理负边权的情况，我们讨论的范围也限定在**无负边权的有限有向图**中。

下面用一个例子直观表示我们的问题。

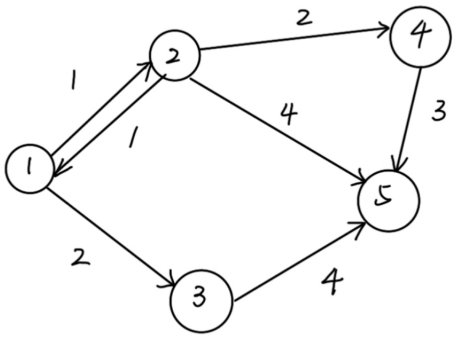

图 1.1: 问题描述: 示例说明

这个例子中，我们要求节点 1 到节点 5 的第 k 短路径。

- k=1 时，即求最短路径，为 1→2→5，长度为 5:
- k=2 时，即求次短路径，为 1→2→4→5，长度为 6:
- k=3 时，第三短的路径为 1→3→5，长度为 6:
- k=4 时，第四短的路径为 1→2→1→2→5，长度为 7:
- .....

不难发现，这个例子中由于存在 1→2→1 的环，节点 1 到节点 5 有无数条路径，因此 k 无论为多大都有解。而如果将 2→1 的回边断开，节点 1 到节点 5 有且仅有上面答案中的前三条路径，当 $k \geq 4$ 时期望输出 “No Solution”。此外，路径的不同是由经过点的集合不同来决定的，而不是路径权值，上面例子中 $k = 2$、$k = 3$ 时结果均为 6 能说明这个问题。

输入输出格式

接受输入：第一行五个整数数 n,m,s,t,k, 分别为节点数目、边数目、起点编号、终点编号、表示要求的 k；接下来 m 行，每行三个整数 u，v，t，表示图中存在一条 $u \to v$、权值为 t 的有向边。

接受输出: 一行整数 l，表示第 k 短路径长度。

2 算法实现:A* 搜索

2.1 算法前瞻

用优先队列优化的 Dijstra 算法不断重复以下的过程：每次出队当前 d 值最小的未标记节点，该次目标并对该节点的所有直接后继节点进行扩展更新、将入队。

在求解最短路算法过程中，我们的目标是让终点节点 t 尽快首次出队 (此时 d[t] 即为最短路结果)，因此入队条件为 d 值可以紧缩，这样的操作有利于减少队列空间。假若我们对入队的条件更改为对任意节点扩展时进行，有一个较为显然的结论是：

当一个点第 k 次出队的时候，此时路径长度就是 s 到它的第 k 短路。

利用这个结论，我们可以直接得出 k 短路结果，但由于状态空间过大，队列空间可能溢出。我们需要设计一种方案，使得相对接近终点的状态优先拓展，这就是 A* 算法设计的灵感。

2.2 算法思想

A* 算法的本质是对上述结论的利用，在不影响正确性的前提下利用估值函数减少状态空间，从而达到空间和时间上的优化。

作为常见的启发式搜索算法，A* 算法定义了一个对当前状态 x 的估价函数:

$$f(x) = g(x) + h(x)$$

其中 $g(x)$ 为从初始状态到达当前状态的实际代价，$h(x)$ 为从当前状态到达目标状态的最佳路径的估计代价。在求解 k 短路问题时，令 $h(x)$ 为从当前结点到达终点 t 的最短路径长度。**可以通过在反向图上对结点 t 跑单源最短路预处理出对每个结点的这个值**。

2.3 算法实现

与 Dijkstra 算法类似，每次取出 $f(x)$ 最优的状态 x，扩展其所有子状态，可以用优先队列来维护这个值。

由于设计的距离函数和估价函数，对于每个状态需要记录两个值，为当前到达的结点 x 和已经走过的距离 $g(x)$，将这种状态记为 $(x, g(x))$。

开始我们将初始状态 (s,0) 加入优先队列。每次我们取出估价函数 $f(x) = g(x) + h(x)$ 最小的一个状态，枚举该状态到达的结点 x 的所有出边，将对应的子状态加入优先队列。当我们访问到一个结点第 k 次时，对应的状态的 $g(x)$ 就是从 x 到该结点的第 k 短路。

优化：由于只需要求出从初始结点到目标结点的第 k 短路，所以已经取出的状态到达一个结点的次数大于 k 次时，可以不扩展其子状态。因为之前 k 次已经形成了 k 条合法路径，当前状态不会影响到最后的答案。

下面给出伪代码，具体代码请参考项目文件。

Algorithm 1 A* 算法求解 k 短路问题

```
1:  procedure Main(n, m, s, t, k)
2:      初始化图和相关数组: 建立正向图和反向图
3:      在反向图上运行 Dijkstra 算法，初始化 H[t] = 0，其余 H[i] = ∞。
4:      while 优先队列 Q 不为空 do
5:          弹出 Q 中顶点 x 与最小估计代价 d。
6:          if x 未访问 then
7:              标记 x 已访问。
8:              for all x 的邻接点 j do
9:                  if 通过 x 到 j 的代价小于已知代价 then
10:                     更新 H[j]。
11:                     将 (j, d + w) 入队至 Q。
12:                 end if
13:             end for
14:         end if
15:     end while
16:     在正向图上运行 A* 算法，初始化源点 s。
17:     while 优先队列 q 不为空 do
18:         弹出 q 中顶点 x 与当前代价 d。
19:         if x 是终点 t 且是第 k 次访问 then
20:             输出 d 为第 k 短路径长度。
21:             return
22:         end if
23:         for all x 的邻接点 j do
24:             if j 的访问次数小于 k then
25:                 将 (j, d + w) 入队至 q。
26:             end if
27:         end for
28:     end while
29: end procedure
```

2.4 复杂度分析

当图的形态是一个 n 元环的时候，该算法每次扩展到一个节点后必须完整遍历完整个环才能再次到达该节点，因此复杂度最坏是 $O(nklogn)$ 的。

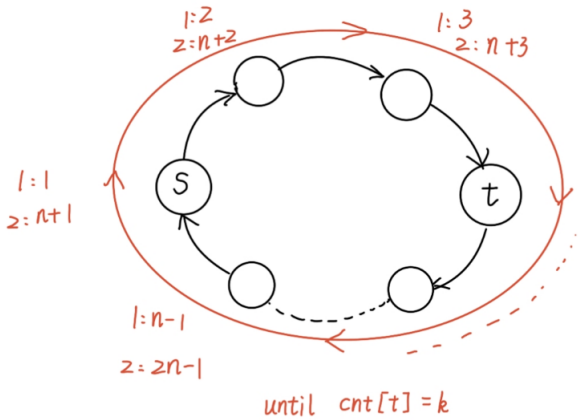

图 2.2: 最坏复杂度情形:n 元环

3 算法实现: 最短路径树 + 可持久化可并堆优化

3.1 前置知识: 可持久化可并堆

左偏树

堆在先前的数据结构中已经学习过，本质为一棵特殊的完全二叉树，满足左右子节点的值都大于/小于父节点。此处我们需要考虑的数据结构为**左偏树**，它在堆的基础上添加了一些限制，也因此拥有额外的性质:

- 对于一棵二叉树，我们定义外节点为左儿子或右儿子为空的节点，定义一个外节点的 dist 为 1，一个不是外节点的节点 dist 为其到子树中最近的外节点的距离加 +1。空节点的 dist 为 0。
- 一棵有 n 个节点的二叉树，根的 dist 不超过 $\lceil \log(n+1) \rceil$，因为一棵根的 dist 为 x 的二叉树至少有 x-1 层是满二叉树，那么就至少有 $2^x - 1$ 个节点。注意这个性质是所有二叉树都具有的，并不是左偏树所特有的。
- 左偏树是一棵二叉树，它不仅具有堆的性质，并且是「左偏」的：每个节点左儿子的 dist 都大于等于右儿子的 dist。

  因此，左偏树每个节点的 dist 都等于其右儿子的 dist + 1。**左偏树是一种特殊的可合并的堆**。

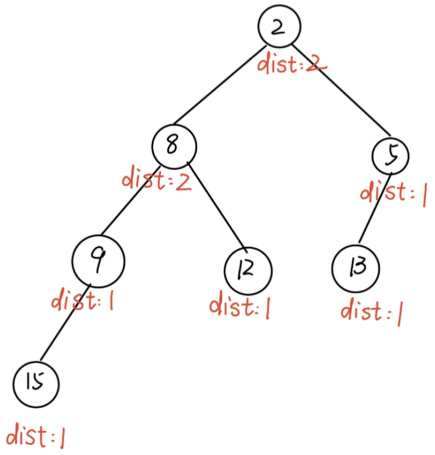

图 3.3: 左偏树

- 左偏树的插入删除操作与堆类似，主要为自底向上的调整节点位置，这里不再考虑，主要讲一下两个左偏树的合并操作 (我们以小根堆为例):
  - 实现流程: 合并两个堆时，由于要满足堆性质，先取值较小（为了方便，本文讨论小根堆）的那个根作为合并后堆的根节点，然后将这个根的左儿子作为合并后堆的左儿子，递归地合并其右儿子与另一个堆，作为合并后的堆的右儿子。为了满足左偏性质，合并后若左儿子的 dist 小于右儿子的 dist，就交换两个儿子。
  - 复杂度分析: 由于左偏性质，每递归一层，其中一个堆根节点的 dist 就会减小 1，而一棵有 n 个节点的二叉树，根的 dist 不超过 $\lceil \log(n+1) \rceil$，所以合并两个大小分别为 n 和 m 的堆复杂度是 $O(\log n + \log m)$。

可持久化左偏树

可持久化要求保留历史信息，使得之后能够访问之前的版本。要将左偏树可持久化，就要将其沿途修改的路径复制一遍。

所以可持久化左偏树的合并过程是这样的:

- 如果 x,y 中有结点为空，返回 x+y。
- 选择 x,y 两结点中权值更小的结点，新建该结点的一个复制 p，作为合并后左偏树的根。
- 递归合并 p 的右子树与 y，将合并后的根节点作为 p 的右儿子。
- 维护以 p 为根的左偏树的左偏性质，维护其 dist 值，返回 p。

由于左偏树一次最多只会修改并新建 $O(\log n)$ 个结点，设操作次数为 m，则可持久化左偏树的时间复杂度和空间复杂度均为 $O(m \log n)$。

3.2 前置知识: 最短路径树

以一个节点为根，该根节点到其他所有点的最短路对应路径保留、其它边删除形成的一棵树。用单源最短路径算法时可以求出源点到其它所有点的最短路径长度，求具体由哪些边构成的路径可以用 dfs 解决: 如果 $dis[v] = dis[u] + w$, 说明这条边可以充当最短路径树上的边。

需要注意的是最短路径树可能不止一个 (两点之间最短路径可能不止一条)，无法用一棵最短路径树表示两点之间所有的最短路径，但我们只需要任意一棵最短路径树即可。此处引用 CSDN 博主的图来举例说明最短路径树的形式。

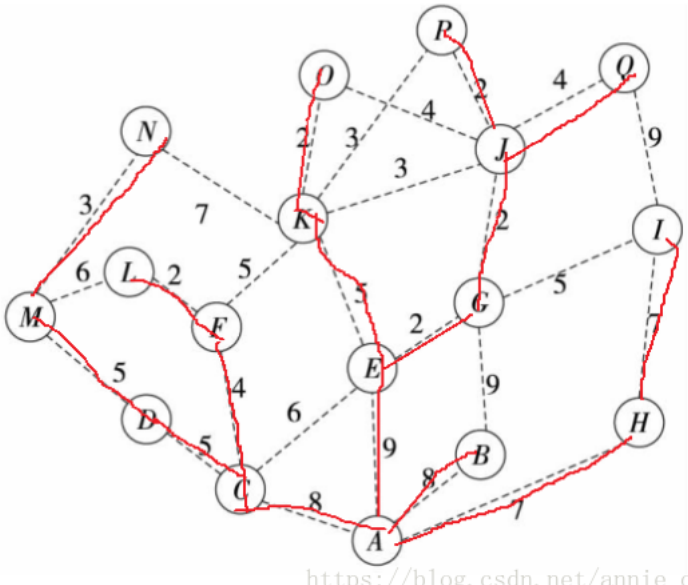

图 3.4: 最短路径树

3.3 算法前瞻

在反向图上从 t 开始跑最短路，设在原图上结点 x 到 t 的最短路长度为 $dist_x$，建出任意一棵以 t 为根的最短路树 T(即我们求出的是任意一个点到终点 t 形成的最短路径树)。

性质

设一条从 s 到 t 的路径经过的边集为 P，去掉 P 中与 T 的交集得到 P'(即 P' 为路径 P 去掉非树边之后的结果)。

P' 有如下性质：

1. 对于一条不在 T 上的边 e，其为从 u 到 v 的一条边，边权为 w，定义其代价 (即为选择该边后路径长度的增加量):

$$\Delta e = dist_v + w - dist_u$$

则路径 P 的长度:

$$L_P = dist_s + \sum_{e \in P'} \Delta e$$

2. 将 P 和 P' 中的所有边按照从 s 到 t 所经过的顺序依次排列，记 u_e 和 v_e 分别为边 e 的前端点（靠近起点）和后端点（靠近终点）。则对于 P' 中相邻的两条边 e_1,e_2(e_1 在前，注意 e_1 与 e_2 在原图路径中不一定相邻)，有 u_{e_2} 与 v_{e_1} 两点要么相等，要么 u_{e_2} 为 v_{e_1} 在 T 上的祖先。因为在原路径集合 P 中 e_1,e_2 要么直接相连 (对应两点相等情况) 或中间用树边相连 (对应在 T 上为祖辈节点的条件)。

3. 对于一个确定存在的 P'，有且仅有一条 $s \to t$ 的路径 P 与之对应。因为最短路树上两点只有一条只经过树边的路径，而 P' 中其他连续的路径段也是确定的。

问题转化

1. 性质 1 告诉我们知道集合 P' 后，如何求出 L_P 的值。

2. 性质 2 告诉我们所有 P' 一定满足的条件，所有满足这个条件的边集 P' 都是合法的，也就告诉我们生成 P' 的方法。

那么问题转化为：求 L_P 的值第 k 小的满足性质 2 的集合 P'。答案就是以下式子的第 k 小值:

$$L_P = dist_s + \sum_{e \in P'} \Delta e$$

我们需要做的就是不断构造出这 k 个 P'。

3.4 算法思想

由于性质 2，我们可以记录按照从 s 到 t 的顺序排列的最后一条边和 L_P 的值，来表示一个边集 P'。

我们用一个小根堆来维护这样的边集 P'。

初始我们将起点为 1 或 1 在 T 上的祖先的所有的边中 Δe 最小的一条边加入小根堆。

每次取出堆顶的一个边集 S，有两种方法可以生成可能的新边集：

1. 替换 S 中的最后一条边为满足相同条件的 Δe 更大的边 (即 $\Delta e' \geq \Delta e$ 且 $\Delta e'$ 尽可能小，注意 e' 为非树边)。

2. 在最后一条边后接上一条边，设 x 为 S 中最后一条边的终点，由性质 2 可得这条边需要满足其起点为 x 或 x 在 T 上的祖先 (即在 x 后面新接上一条最短路树上祖先方向出去的 Δe 值最小的边 e)。

将生成的新边集也加入小根堆。重复以上操作 k-1 次后求出的就是从 s 到 t 的第 k 短路。

维护信息: 如何快速构造新路径?

目前我们实现了通过现有的一条 $s \to t$ 的路径得出另一条更长的路径，而上述思想中如何快速获取需求的边仍然是一个挑战。我们通过维护以下信息来解决:

对于每个结点 x，我们将以其为起点的边的 Δe 建成一个小根堆。为了方便查找一个结点 x 与 x 在 T 上的祖先在小根堆上的信息，我们将这些信息合并在一个编号为 x 的小根堆上。回顾以上生成新边集的方法，我们发现只要我们把紧接着可能的下一个边集加入小根堆，并保证这种生成方法可以覆盖所有可能的边集即可。记录最后选择的一条边在堆上对应的结点 t，有更优的方法生成新的边集:

1. 替换 S 中的最后一条边为 t 在堆上的左右儿子对应的边。
2. 在最后一条边后接上一条新的边，设 x 为 S 中最后一条边的终点，则接上编号为 x 的小根堆的堆顶结点对应的边。

用这种方法，每次生成新的边集只会扩展出最多三个结点，小根堆中的结点总数是 $O(n+k)$。

所以此算法的瓶颈在合并一个结点与其在 T 上的祖先的信息，如果使用朴素的二叉堆，时间复杂度为 $O(nm \log m)$，空间复杂度为 $O(nm)$ ；如果使用可并堆，每次仍然需要复制堆中的全部结点，时间复杂度同样无法承受。故考虑使用可持久化可并堆优化合并一个结点与其在 T 上的祖先的信息，每次将一个结点与其在 T 上的父亲合并，这样在求出一个结点对应的堆时，无需复制结点且之后其父亲结点对应的堆仍然可以正常访问。

3.5 算法实现

具体的算法实现流程归纳为如下:

1. 在反图上跑 Dijkstra, 再通过 dfs 构造最短路径树;
2. 构造可持久化左偏树:
   - 对于每个节点都扫一遍邻边（除树边）e，然后将其 Δe 值与另一端点编号一并插入当前点的左偏树中;
   - 然后向树边祖先方向将堆合并。
3. 构造满足大小要求的前 k 个 P',
   - 取出堆顶;
   - 向左偏树节点儿子拓展;
   - 向对应边结束点的左偏树的根拓展。

在堆中取出的第 k 个 P' 即为答案对应方案，按照公式计算结果即可。

下面给出伪代码，具体代码请见项目文件:

Algorithm 2 最短路径树 + 可持久化左偏树求解 k 短路问题

```
procedure MAIN(G = (V, E), s, t, k)
    在反图 G' 上运行 Dijkstra 算法，初始化 dist[t] = 0，其余 dist[v] = ∞ 对所有 v ∈ V
    while 存在未处理的节点 do
        从优先队列中提取节点 u 与最小 dist[u]
        for all u 的邻接节点 v do
            if dist[v] > dist[u] + w(u, v) then
                更新 dist[v] ← dist[u] + w(u, v)
            end if
        end for
    end while
    通过 DFS 从 t 构建最短路径树 T
    初始化可持久化左偏树 ST 并为每个节点 v ∈ V 创建一个左偏树的根节点
    for all 节点 x 且 x ≠ t do
        for all 边 e = (x, y) ∉ T do
            计算 Δe = dist[y] + w(e) − dist[x]
            将 Δe 和边 e 插入到 x 的左偏树中
        end for
        if 存在 fa[x] 在 T 中 then
            将 x 的左偏树与 fa[x] 的左偏树合并
        end if
    end for
    初始化优先队列 Q 并将其用于维护可持久化左偏树的根节点
    将 s 的左偏树的根节点和 dist[s] 入队至 Q
    cnt ← 0
    while Q 不为空 do
        提取 Q 中顶点对应的左偏树节点 u 和路径长度 d
        if cnt = k − 1 then
            输出 d 为第 k 短路径的长度
            return
        end if
        cnt ← cnt + 1
        向左偏树节点 u 的左右子节点拓展，并更新 Q
    end while
end procedure
```

3.6 复杂度分析

如上文中对可持久化左偏树的复杂度分析那样，单次合并操作为 log 级，总的时间复杂度为 $O((n+m)\log m + k\log k)$，空间复杂度为 $O(m + n\log m + k)$。

需要注意的是，如上文所言，最终询问时不需要可并堆的合并操作。询问时使用优先队列维护可并堆的根，对于可并堆堆顶的删除，直接将其左右儿子加入优先队列中，就只需要 $O(k)$ 而非 $O(k\log m)$ 的空间。

附录 A 实验环境

源码链接 (含数据)

A.1 本机硬件参数

CPU 型号	12th Gen Intel(R) Core(TM) i7-12700H
CPU 核数	20
CPU 主频	2.3GHz
CPU 缓存 (L1)	1.2MB
CPU 缓存 (L2)	11.5MB
CPU 缓存 (L3)	24.0MB
内存容量	16384MB RAM

表 1: 硬件参数

A.2 系统配置与编译器信息

为了更加直观地体现出大数据规模下两个算法的差距，编译选项设定为-O0, 即不进行任何优化。

系统版本	Windows 11 家庭中文版 (23H2)
操作系统版本	22631.3296
编译器版本	TDM-GCC 4.9.2 64-bit Release
编译选项	-std=c++11 -O0

表 2: Windows 环境

A.3 实验数据

数据来源为并查集随机构造一棵树 + 树上随机连边，其中边权为随机数取得。详细数据、数据生成器见附录源码链接。

为了保证生成的图存在 k 短路，我们将设定 $m \geq n^2$(即一个较为稠密的图)。正确性在数据规模较小时已经得到验证 (如报告首页底部的样例)，较大数据规模主要判断能否在有限的时间内得到结果。