打破排序障碍的最短路算法

2025-2026 学年秋冬学期计算经济学讨论班

郑涵文

zhwen@zju.edu.cn

2025 年 9 月 25 日

Introduction

单源最短路径是一个非常经典的问题，我们最常使用的算法是 Dijkstra 算法．由于它依赖于对所有（候选）节点的整体排序，所以其时间复杂度至少为 $O(n\log n)$．Dijkstra 在 2024 年被证明具有普遍最优性：如果我们要求最短路算法按距离顺序输出各个节点，那么 Dijkstra 是最优的算法．但如果我们只是求出每个节点离原点的距离，那么结果就不一样了．

本文作者通过一个不进行排序的确定性算法，能够把时间复杂度降低到 $O(m\log^{2/3}(n))$，也就是说在稀疏图（n, m 同阶）上它比 Dijkstra 更优．

回顾

算法整体上使用了递归分治的技巧，在处理过程中结合了 Dijkstra 和 Bellman-Ford 两个经典算法.

Dijkstra

通过优先队列，它每次提取一个距离源点最小的顶点 u，并松弛 u 的各个出边. 时间复杂度至少为 $O(n\log n)$.

Bellman-Ford

它基于动态规划的思想，迭代地松弛所有边 $n-1$ 次. 对于最多经过 k 条边的最短路径，Bellman-Ford 算法可以在 $O(mk)$ 时间内求解，并且无需任何排序.

前沿集

在 Dijkstra 算法执行过程中的任意时刻，优先队列（堆）维护了一个顶点“前沿” S，使得如果某个顶点 u 是“不完整的”——即当前距离估计值 $\hat{d}[u]$ 仍大于真实距离 $d(u)$——那么最短 s-u 路径必须经过某个完整顶点 $v \in S$．这种情况下，我们称 u“依赖”于 $v \in S$．（需要注意的是，S 中的顶点并不保证都是完整的．）Dijkstra 算法简单地选择 S 中离源点最近的顶点（该顶点必然是完整的），然后松弛从该顶点出发的所有边．

Dijkstra 运行时间的瓶颈基于此事实：有时前沿可能包含 $\Theta(n)$ 个顶点．由于我们需要不断选择离源点最近的顶点，这本质上需要维护大量顶点之间的全序关系，因此无法突破 $\Omega(n \log n)$ 的排序障碍．作者最核心的想法是一种减少前沿大小的方法．

前沿集

假设我们想计算所有小于某个上界 B 的距离．设 $\tilde{U}$ 为满足 $d(u) < B$ 且最短 s-u 路径经过 S 中某个顶点的顶点集合．我们有可能将前沿大小 $|S|$ 限制在 $|\tilde{U}|/\log^{\Omega(1)}(n)$ 以内，即感兴趣顶点数量的 $1/\log^{\Omega(1)}(n)$ 倍．给定参数 $k = \log^{\Omega(1)}(n)$，有两种可能情况：

1. 如果 $|\tilde{U}| > k|S|$，那么我们的前沿大小已经是 $|\tilde{U}|/k$;
2. 否则，假设 $|\tilde{U}| \leq k|S|$．通过从 S 中的顶点运行 k 步 Bellman-Ford 算法，每个最短路径包含 $< k$ 个 $\tilde{U}$ 中顶点的顶点 $u \in \tilde{U}$ 都会变成完整的．否则，u 所依赖的顶点 $v \in S$ 必然有一个以 v 为根的最短路径树，该树包含 $\geq k$ 个 $\tilde{U}$ 中的顶点，因此我们可以将前沿 S 缩减为“关键点”集合，每个关键点都有一个大小 $\geq k$ 的最短路径树，而显然这类关键点的数量不超过 $|\tilde{U}|/k$，因为每个 $\tilde{U}$ 中的节点都被唯一的最短路径树的树根控制．

前沿集

算法基于上述思想，但不同于 Dijkstra 类方法（其前沿是动态的且难以处理），我们采用一种分治过程，该过程包含 $(\log n)/t$ 个层级，每个层级包含一个前沿顶点集合和一个上界 B. 如果采用朴素实现，每个前沿顶点仍需花费 $\Theta(t)$ 时间，总运行时间仍为每个顶点 $\Theta(\log n)$. 然而，我们可以在每个层级上应用上述前沿缩减方法，使得 $\Theta(t)$ 的工作量仅适用于关键点 (约占前沿顶点数量的 $1/\log^{\Omega(1)}(n)$). 因此，每个顶点的运行时间减少到 $\log n/\log^{\Omega(1)}(n)$，这是一个显著的加速.

模型、符号、定义

有向图 $G=(V,E)$，非负边权 $w: E\to\mathbb{R}_{\geq 0}$，源点 $s\in V$，$n=|V|, m=|E|$. 目标是计算所有顶点 v 的最短路距离 $d(v)$. 使用 $\hat{d[v]}$ 表示算法运行到目前为止，$d(v)$ 的估计值.

图是常数度数的. 可以证明任何图 $G=(V,E), n=|V|, m=|E|$ 都能转化为每个节点出度和入度均不大于 2 的图 $G'=(V',E')$，其中 $|V'|,|E'|=O(m)$，并且不影响原图的最短路求解.

所有路径都有不同的长度. 或者说，任意两条路径一定能进行比较，并排出先后关系.（首先比较路径长度，再比较路径上节点个数，再比较路径上的节点字典序...）这样做是为了使得每条路径都不一样，方便处理.

浙江大学 ZHEJIANG UNIVERSITY

算法思想

令 $k := \lfloor \log^{1/3}(n) \rfloor$ 和 $t := \lfloor \log^{2/3}(n) \rfloor$ 为我们算法中的两个参数．我们的主要思想基于对顶点集进行分治．我们希望将一个顶点集 U 划分为 2^t 个大小相近的子集 $U = U_1 \cup U_2 \cup \cdots \cup U_{2^t}$，其中较早子集中的顶点具有较小的距离，然后递归地对每个 U_i 进行划分．通过这种方式，子问题的规模在大约 $(\log n)/t$ 层递归后会缩小到单个顶点．

假设在算法的某个阶段，对于每个满足 $d(u) < b$ 的顶点 u，u 是完整的，并且我们已经松弛了从 u 出发的所有边．现在，我们想要找到满足 $d(v) \geq b$ 的顶点 v 的真实距离．为了避免在优先队列中每个顶点花费 $\Theta(\log n)$ 的时间，考虑"前沿" S，它包含所有当前满足 $b \leq d[v] < B$ 的顶点 v（其中 B 是某个上界，且不对这些顶点进行排序）．我们可以看出，每个不完整顶点 v'（满足 $b \leq d(v') < B$）的最短路径必须经过 S 中的某个完整顶点．因此，要计算每个满足 $b \leq d(v') < B$ 的顶点 v' 的真实距离，只需找到从 S 中顶点出发且不超过 B 的最短路径．我们将这个子问题称为有界多源最短路径问题，并设计了一个高效算法来解决它．以下引理总结了我们的 BMSSP 算法所实现的效果．

算法思想

引理 1

给定一个整数层级 $l \in [0, \lceil \log(n)/t \rceil]$，一个顶点集合 S 满足 $|S| \leq 2^{lt}$，以及一个上界 $B > \max_{x \in S} \hat{d}[x]$. 假设对于每个不完整顶点 v（满足 $d(v) < B$），其最短路径都经过某个完整顶点 $u \in S$.
那么，我们有一个子程序 BMSSP(l, B, S)（算法 3），它能在 $O((kl + tl/k + t)|U|)$ 的时间内输出一个新的边界 $B' \leq B$ 和一个顶点集合 U，该集合 U 包含所有满足 $d(v) < B'$ 且其最短路径经过 S 中某顶点的顶点 v. 在该子程序结束时，U 中的顶点都是完整的.

寻找关键点

继续我们一开始提到的减少前沿的数量的思想，我们需要设计一个算法去找到关键点．寻找关键点的思想如下：我们执行 k 步松弛操作（以 B 为上界）．在此之后，如果满足 $b \le d(v) < B$ 的最短 s-v 路径最多经过 k 个满足 $b \le d(w) < B$ 的顶点 w，那么 v 就已经是完整的．观察到，从 S 出发的大型最短路径树（包含至少 k 个顶点且根节点在 S 中）的数量最多为 $|\tilde{U}|/k$，其中 $\tilde{U}$ 是所有满足 $d(v) < B$ 且其最短路径经过 S 中某个完整顶点的顶点 v 的集合．因此，在递归调用中只需考虑这类最短路径树的根节点，它们被称为“关键点”．

引理 2

假设给定一个上界 B 和一个顶点集合 S．假设对于每个不完整顶点 v（满足 $d(v) < B$），其最短路径都经过某个完整顶点 $u \in S$．令 $\tilde{U}$ 为包含所有满足 $d(v) < B$ 且其最短路径经过 S 中某顶点的顶点 v 的集合．子程序 FindPivots(B, S) 会找到一个大小为 $O(k|S|)$ 的集合 $W \subseteq \tilde{U}$ 和一个大小最多为 $|W|/k$ 的关键点集 $P \subseteq S$，使得对于每个顶点 $x \in \tilde{U}$，至少满足以下两个条件之一：

1. 在子程序结束时，$x \in W$ 且 x 是完整的；
2. 到 x 的最短路径经过某个完整顶点 $y \in P$.

寻找关键点

Algorithm 1 Finding Pivots

function FINDPIVOTS(B, S)

- **requirement:** for every incomplete vertex v with $d(v) < B$, the shortest path to v visits some complete vertex in S
- **returns:** sets P, W satisfying the conditions in Lemma 3.2

$W \leftarrow S$

$W_0 \leftarrow S$

for $i \leftarrow 1$ to k **do** ▷ Relax for k steps

$W_i \leftarrow \emptyset$

for all edges (u, v) with $u \in W_{i-1}$ **do**

if $\widehat{d}[u] + w_{uv} \leq \widehat{d}[v]$ **then**

$\widehat{d}[v] \leftarrow \widehat{d}[u] + w_{uv}$

if $\widehat{d}[u] + w_{uv} < B$ **then**

$W_i \leftarrow W_i \cup \{v\}$

$W \leftarrow W \cup W_i$

if $|W| > k|S|$ **then**

$P \leftarrow S$

return P, W

$F \leftarrow \{(u, v) \in E : u, v \in W, \widehat{d}[v] = \widehat{d}[u] + w_{uv}\}$ ▷ F is a directed forest under Assumption 2.1

$P \leftarrow \{u \in S : u$ is a root of a tree with $\geq k$ vertices in $F\}$ 只有≥k的树还有伸展机会

return P, W

数据结构

为了能够方便我们将大问题分割成若干子问题以进行递归求解，引入一个数据结构进行帮助.

引理 3

给定最多 N 个待插入的键值对、一个整数参数 M 以及所有涉及值的上界 B，存在一个数据结构支持以下操作：

1. 插入：以均摊时间复杂度 $O(\max 1, \log(N/M))$ 插入一个键值对. 如果键已存在，则更新其值.
2. 批量前插：以均摊时间复杂度 $O(L \cdot \max 1, \log(L/M))$ 插入 L 个键值对，其中每个值都小于当前数据结构中的任何值. 如果存在多个相同键的配对，则保留值最小的那个.
3. 拉取：以均摊时间复杂度 $O(|S'|)$ 返回一个键的子集 S'，其中 $|S'| \leq M$，包含最小的 $|S'|$ 个值，以及一个上界 x，用于将 S' 与数据结构中剩余的值分开. 具体而言，如果没有剩余值，x 应为 B；否则，x 应满足 $\max(S') < x \leq \min(D)$，其中 D 是拉取操作后数据结构中的元素集合.

浙江大学 ZHEJIANG UNIVERSITY

具体算法

回顾一下，主算法在顶层调用 BMSSP 时使用的参数是 $l = \lceil (\log n)/t \rceil$，$S = \{s\}$，$B = \infty$．在 base case 即最底层递归，$l = 0$ 时，S 是一个单例集合 $\{x\}$ 且 x 是完整的．我们运行一个从 x 出发的微型 Dijkstra 算法（算法 2），以找到那些满足 $d(v) < B$ 且到 v 的最短路径经过 x 的、离 x 最近的顶点 v，直到找到 $k+1$ 个这样的顶点或者找不到更多顶点为止．令 U_0 为这些顶点的集合．如果我们没有找到 $k+1$ 个顶点，则返回 $B' \leftarrow B$，$U \leftarrow U_0$．否则，返回 $B' \leftarrow \max_{v \in U_0} d(v)$，$U \leftarrow \{v \in U_0 : d(v) < B'\}$．

具体算法

整个算法如下：

function BMSSP(l, B, S)

• **requirement 1:** $|S| \leq 2^{lt}$

• **requirement 2:** for every incomplete vertex x with $d(x) < B$, the shortest path to x visits some complete vertex $y \in S$

• **returns 1:** a boundary $B' \leq B$

• **returns 2:** a set U

  if $l = 0$ **then**

    return $B', U \leftarrow \textsc{BaseCase}(B, S)$

  $P, W \leftarrow \textsc{FindPivots}(B, S)$

  $\mathcal{D}.\textsc{Initialize}(M, B)$ with $M = 2^{(l-1)t}$ ▷ $\mathcal{D}$ is an instance of Lemma 3.3

  $\mathcal{D}.\textsc{Insert}(\langle x, \widehat{d}[x] \rangle)$ **for** $x \in P$

  $i \leftarrow 0$; $B'_0 \leftarrow \min_{x \in P} \widehat{d}[x]$; $U \leftarrow \emptyset$ ▷ If $P = \emptyset$ set $B'_0 \leftarrow B$

  while $|U| < k2^{lt}$ and $\mathcal{D}$ is non-empty **do**

    $i \leftarrow i + 1$

    $B_i, S_i \leftarrow \mathcal{D}.\textsc{Pull}()$

    $B'_i, U_i \leftarrow$ BMSSP$(l - 1, B_i, S_i)$

    $U \leftarrow U \cup U_i$

    $K \leftarrow \emptyset$

    for edge $e = (u, v)$ where $u \in U_i$ **do**

      if $\widehat{d}[u] + w_{uv} \leq \widehat{d}[v]$ **then**

        $\widehat{d}[v] \leftarrow \widehat{d}[u] + w_{uv}$

        if $\widehat{d}[u] + w_{uv} \in [B_i, B)$ **then**

          $\mathcal{D}.\textsc{Insert}(\langle v, \widehat{d}[u] + w_{uv} \rangle)$

        else if $\widehat{d}[u] + w_{uv} \in [B'_i, B_i)$ **then**

          $K \leftarrow K \cup \{\langle v, \widehat{d}[u] + w_{uv} \rangle\}$

    $\mathcal{D}.\textsc{BatchPrepend}(K \cup \{\langle x, \widehat{d}[x] \rangle : x \in S_i \text{ and } \widehat{d}[x] \in [B'_i, B_i)\})$

  return $B' \leftarrow \min\{B'_i, B\}$; $U \leftarrow U \cup \{x \in W : \widehat{d}[x] < B'\}$

正确性

对于此算法的正确性可以根据归纳法大致理解：在 base case 中，当 $l=0$ 时，S 仅包含一个顶点 x，从 x 开始的类 Dijkstra 算法完成任务．
否则，我们首先将 S 缩减为一个更小的关键点集 P 插入到 D 中，其中一些顶点变为完整的并被加入 W．引理 3.2 确保任何剩余的不完整顶点 v 的最短路径都经过 D 中的某个完整顶点．对于任何界限 $B_i \leq B$，如果 $d(v) < B_i$，则到 v 的最短路径也必须经过 D 中某个满足 $d(u) < B_i$ 的完整顶点 u．因此，我们可以调用子过程 BMSSP，参数为 B_i 和 S_i．
根据归纳假设，每次算法 3 第 11 行的递归调用返回时，U_i 中的顶点都是完整的．在算法松弛从 $u \in U_i$ 出发的边，并将所有更新的出边邻居 x（满足 $d_b[x] \in [B'_i, B)$）插入 D 后，再次地，任何剩余的不完整顶点 v 现在都经过 D 中的某个完整顶点．
最后，加入 W 中的完整顶点后，U 包含了所有所需的顶点，且这些顶点都是完整的．
所以在整个算法调用结束后，所有节点都是完整的．

运行时间

对于算法运行时间也能进行一个简略的分析，运行时间主要由 FindPivots 的调用和数据结构 D 内部的开销所主导．我们已经知道了 FindPivots 过程在节点 x 上的运行时间受 $O(|U_x|k)$ 限制，因此 T 树某一深度上所有节点的总运行时间为 $O(nk)$．对所有深度求和，我们得到 $O(nk \cdot (\log n)/t)$．

对于数据结构 D 能得到一个主导项 $O((n\log n)/k)$，可以发现 $k := \lfloor \log^{1/3}(n) \rfloor$ 和 $t := \lfloor \log^{2/3}(n) \rfloor$ 时有最小的总复杂度 $O(n \log^{2/3} n)$．