顶点故障下的连通性查询：并非最优，但切实可行

Evangelos Kosinas*

2023年9月6日

arXiv:2305.01756v2 [cs.DS] 1 Sep 2023

摘要

我们再次审视这样一个问题：如何设计预言机，以便在存在顶点故障的情况下回答无向图中的连通性查询。具体而言，给定一个具有 n 个顶点和 m 条边的无向图 G，以及一个整数 $d_\star \ll n$，目标是对该图进行预处理，以构造数据结构 $\mathcal{D}$，使得给定一个满足 $|F| = d \leq d_\star$ 的顶点集合 F 后，我们可以从 $\mathcal{D}$ 导出一个预言机，高效地回答如下查询："x 和 y 在 $G \setminus F$ 中是否连通？"。最近，Long 和 Saranurak（FOCS 2022）为这个问题提供了一种解决方案，在预处理时间、空间使用量、更新时间和查询时间方面几乎达到了最优。然而，他们的方案非常复杂，似乎很难高效实现。此外，当 $d_\star$ 为常数时，该方案并未解决这一问题的复杂度。在本文中，我们为这个问题提供了一种简单得多的解决方案，仅使用教科书式数据结构。我们的算法是确定性的，其预处理时间和空间复杂度为 $O(d_\star m \log\, n)$，更新时间为 $O(d^4 \log\, n)$，查询时间为 $O(d)$。考虑到我们方法的简单性，这些界限与此前的最佳结果相比颇具竞争力。事实上，如果假设 $d_\star$ 为常数（$d_\star \geq 4$），那么我们的算法还能实现一些权衡，在某些方面提升技术水平。最后，我们提供的数据结构对于 $d_\star$ 具有灵活性：它可以适用于增加和减少，其时间和空间开销几乎与 $d_\star$ 的变化量及图的大小成正比。

*希腊约阿尼纳大学。电子邮件：ekosinas@cs.uoi.gr。本研究工作得到希腊研究与创新基金会（HFRI）HFRI 博士奖学金第三轮申请的资助（奖学金编号：6547）。

1 引言

本文研究以下问题。给定一个具有 n 个顶点和 m 条边的无向图 G，以及一个固定整数 $d_\star$（$d_\star \ll n$），目标是构造数据结构 $\mathcal{D}$，以便在至多 $d_\star$ 个顶点发生故障的情况下回答连通性查询。更准确地说，给定一个满足 $|F| \leq d_\star$ 的顶点集合 F，我们必须能够从 $\mathcal{D}$ 高效地导出一个预言机，以便高效回答如下查询："顶点 x 和y 在 $G \setminus F$ 中是否连通？"。在这个问题中，我们希望同时优化以下参数：(1) $\mathcal{D}$ 的构造时间（预处理时间），(2) $\mathcal{D}$ 的空间使用量，(3) 给定 F 时从 $\mathcal{D}$ 导出预言机所需的时间（更新时间），以及 (4) 在 $G\setminus F$ 中回答连通性查询所需的时间。这个问题有充分的研究动机；十多年来，它一直受到研究人员的关注，而且存在许多有趣的变体。关于该问题及其变体的历史和详情，请参阅 [5] 或 [6] 。

1.1 相关研究

尽管这一问题已得到广泛研究，直到最近 Long 和 Saranurak [6] 才提出了一个近乎最优的解决方案。具体而言，他们提出了一种确定性算法，其预处理时间为 $\hat{O}(m) + \tilde{O}(d_\star m)$，空间复杂度为 $O(m\log^* n)$，更新时间为 $\hat{O}(d^2)$，查询时间为 $O(d)$。[1] 这改进了此前由 Duan 和 Pettie [5], 提出的最佳确定性解决方案；该方案的预处理时间为 $O(mn\log\ n)$，空间复杂度为 $O(d_\star m \log\ n)$，更新时间为 $O(d^3\log^3 n)$，查询时间为 $O(d)$。我们注意到，这一问题还有其他解决方案，它们优化了某些参数，同时牺牲了其他参数（例如，在 Pilipczuk 等人 [8], 的解决方案中，更新时间不依赖于 n，但其关于 $d_\star$ 是超指数的，且预处理时间为 $O(mn^2 2^{2^{O(d_\star)}})$）。关于这一问题已知最佳（上界）的更多细节，请参见参考文献 [6] 中的表1。关于已知的（条件）下界总结，请参见 [6] 中的定理 1.2；这些下界证明了 [6] 的最优性。

1.2 我们的贡献

我们提到的时间界是确定性解法中目前已知的最佳结果。实际上，人们更倾向于采用 Long 和 Saranurak [6], 的解法，因为 Duan 和 Pettie [5]的解法的预处理时间为 $O(mn\log\ n)$，对于足够大的图而言，这一耗时可能难以接受。然而，[6] 中的解法非常复杂，似乎很难高效实现。这在理论与实践之间造成了巨大的差距。此外，如果假设 $d_\star$ 对于我们的问题是常数，那么 [6] 时间界中对 n 的（隐含）依赖未必是最优的。我们指出，这是一个涉及多个参数的问题，因此很难同时优化所有参数。

鉴于这是一个基本连通性问题，我们认为，提出一种易于描述和分析、与目前已知的最佳时间界相比表现出色（并在某些方面有所改进）、开辟解决该问题复杂度的新方向，且能够轻松高效实现的解法，十分重要。

在本文中，我们展示了一种恰好具有这些特性的解决方案。我们提出了一种确定性算法，其预处理时间为 $O(d_\star m \log\ n)$，使用的空间为 $O(d_\star m\log\ n)$，

[1]符号 $\hat{O}$ 隐藏了次多项式（即 $n^{o(1)}$）因子，而 $\tilde{O}$ 隐藏了多对数因子。作者在概述中没有明确说明时间界中隐藏的表达式。此外，对于空间复杂度中出现的 $\log^* n$ 函数，其描述是"可以替换为任何增长缓慢的函数"。不过，有一点是明确说明的：所隐藏的次多项式因子比多对数因子更差。我们必须强调，难以给出精确界限，部分原因在于所涉及的函数之间存在各种权衡，部分原因则说明了所使用技术的复杂度。

且具有 $O(d^4 \log n)$ 更新时间，以及 $O(d)$ 查询时间。[2] 可以说，我们的方法是针对这个问题提出过的最简单的方法。以往的解决方案依赖于对原图进行复杂的树分解。相反，我们基本上只依赖一棵深度优先搜索树，并简单分析移除一组顶点后形成的连通分量。事实证明，其中存在足够的结构，可以支持高效求解（见第 3.2 节）。

	预处理	空间	更新	查询
Pilipczuk 等人 [8]	$O(2^{2^{O(d_\star)}} mn^2)$	$O(2^{2^{O(d_\star)}} m)$	–	$O(2^{2^{O(d_\star)}})$
Duan 和 Pettie [5]	$O(mn$ 对数 $n)$	$O(d_\star m$ 对数 $n)$	$O(d^3$ 对数$^3 n)$	$O(d)$
Long 和 Saranurak [6]	$\hat{O}(m) + \tilde{O}(d_\star m)$	$O(m$对数$^* n)$	$\hat{O}(d^2)$	$O(d)$
本文	$O(d_\star m$ 对数 $n)$	$O(d_\star m$ 对数 $n)$	$O(d^4$ 对数 $n)$	$O(d)$

表1：已知最佳确定性界的比较。我们注意到，m 可以替换为 $\bar{m} = $ 最小值$\{m, d_\star n\}$，利用 Nagamochi 和 Ibaraki 的稀疏化 [7]。Pilipczuk 等人的数据结构不支持更新阶段，但可在给定一组（至多 $d_\star$ 个）失效顶点和两个查询顶点后，直接回答查询。

	预处理	空间	更新	查询
Pilipczuk 等人 [8]	$O(mn^2)$	$O(m)$	–	$O(1)$
Duan 和 Pettie [5]	$O(mn$ 对数 $n)$	$O(m$ 对数 $n)$	$O(\log^3 n)$	$O(1)$
Long 和 Saranurak [6]	$\hat{O}(m) + \tilde{O}(m)$	$O(m$对数$^* n)$	$\hat{O}(1)$	$O(1)$
本文	$O(m$ 对数 $n)$	$O(m$对数 $n)$	$O($对数 $n)$	$O(1)$

表 2：当 $d_\star$ 为固定（较小）常数时，已知最佳确定性界的比较。尽管 Pilipczuk 等人的算法在空间和查询时间界方面表现最佳，但其预处理时间非常长。我们的方法具有最佳预处理时间，而且与 [5] 和 [6] 的方法相比，更新时间也更短。此外，我们的空间使用量几乎是线性的。

我们给出的界限与此前的最佳结果相比表现出色，尤其考虑到我们的方法十分简单。（见表1 和2。）事实上，如表1所示，考虑到 Long 和 Saranurak 的算法很难在声称的时间界内实现，我们的方案是实现时的最佳选择。此外，若假设 $d_\star$ 是常数（$d_\star \geq 4$），那么，如表2所示，我们的算法提供了一些权衡，在某些方面改进了技术水平。

最后，我们提供的数据结构在 $d_\star$方面具有灵活性：它可以适应增减，其时间和空间开销几乎与变化的 $d_\star$以及图的规模成正比（见推论3.1）。我们不知道先前的任何解决方案是否具有这一性质。一个自然的问题是，我们能否高效地更新数据结构，使其能够处理更多故障（或更少故障，从而释放一些空间）。据我们所知，我们是第一个注意到问题这一方面的人。

[2]使用更复杂的二维范围空性数据结构（例如 [2] 中的结构），可以降低空间使用量和更新操作时间中的对数因子。

2 预备知识

我们假设读者熟悉标准的图论术语（例如，参见[4]）。我们使用的符号也都是标准的。由于我们研究的是顶点故障下的连通性，因此对于我们的问题，只需考虑以简单图作为输入即可（因为平行边的存在不会影响连通关系）。然而，在更新阶段，我们会构造一个多重图，用于表示移除失效顶点后若干连通分量之间的连通关系（定义3.7）。该图中的平行边是冗余的，但我们使用的算法可能会引入这些平行边，而且在整个过程中持续检查冗余边的代价较高。

假设输入图 G 是连通的也足够。否则，我们可以在 G的每个连通分量上初始化一个数据结构；对于给定的故障集合，更新操作会分发到各个连通分量上的数据结构中，而对于位于 G 不同连通分量中的顶点对，查询结果始终为假。全文中，我们用 G 表示输入图； n 和 m 分别表示其顶点数和边数。对于任意两个整数 x， y，我们使用区间记号 $[x, y]$ 表示集合 $\{x, x+1, \ldots, y\}$。（如果 $x > y$，那么 $[x, y] = \emptyset$。）

2.1 基于深度优先搜索的概念

设 T 是 G的一棵深度优先搜索树，起始顶点为 r [9]。我们用 $p(v)$表示 $v \neq r$ 在 T中的父节点（v是 p的子节点（v））。对于任意两个顶点 u和 v，我们用 $T[u, v]$ 表示 T上从 u到 v的简单树路径。对于任意两个顶点 u和 v，如果树路径 $T[r, u]$经过v，那么我们称 v 是 u 的祖先（等价地，u 是 v的后代）。特别地，一个顶点也被视为自身的祖先（以及后代）。将顶点与其在深度优先搜索期间的访问顺序对应起来非常有用，起始于 $r \leftarrow 1$。因此，如果 v是 u的祖先，则有 $v < u$。对于任意顶点 v，我们用 $T(v)$表示以 v为根的子树，并用 ND(v)表示 v的后代数量（即，ND (v) $= |T(v)|$）。因此，我们有 $T(v) = [v, v + \text{ND}(v) - 1]$，从而可以在常数时间内检查祖先关系。顶点 v 的两个子节点 c 和 c' 称为 v 的连续子节点（按此顺序），若 c' 是 v 中满足 $c' > c$的最小子节点。注意，此时有$T(c) \cup T(c') = [c, c' + \text{ND}(c') - 1]$。

深度优先搜索树 T 有一个极其方便的性质： G 中每条非树边的两个端点，在 T [9]上都互为祖先和后代，因此我们称这些边为回边。我们的整个方法本质上就是利用这一性质；一般的 G 有根生成树并不具备这一性质（除非它们是由深度优先搜索遍历生成的，而且也只有在这种情况下才具备）。为了说明这与我们的目的有何关联，考虑从 T中移除一个顶点 $f \neq r$ 时会发生什么。设 f 在 T 中的子节点为 c_1、 ..... 、 c_k。那么， $T \setminus f$ 的连通分量为 $T(c_1)$、.....、$T(c_k)$，以及 $T(r) \setminus T(f)$。当且仅当存在一条回边从$T(c_i)$出发，并终止于 f的真祖先时，子树 $T(c_i)$（ $i \in \{1, \ldots, k\}$）才在 $G \setminus f$中与图的其余部分连通。现在，这个问题有一个算法上优雅的解法。假设我们已为每个顶点 $v \neq r$计算出 v通过回边与 $T(v)$相连的最低的真祖先。我们将该顶点记为 low (v)。这样，我们只需检查 low (c_i) $< f$，即可确定 $T(c_i)$在 $G \setminus f$中是否与 $T(r) \setminus T(f)$连通。

我们扩展低点的概念，引入适用于任意 $k \in \mathbb{N}$ 的 low_k 点。对于任意顶点 $v \neq r$，其定义递归如下。$\text{low}_1(v)$ 与 low(v) 一致。然后，假设我们已经为某个 $k \in \mathbb{N}$ 定义了 $\text{low}_k(v)$，我们将 $\text{low}_{k+1}(v)$ 定义为满足 $x \in T(v)$ 且 $y < v\} \setminus \{\text{low}_1(v) \ldots, \text{low}_k(v)\}$ 的一条回边 (x, y) 中的 $\{y \mid \exists$ a 的最小值。注意，对于某些 $k \in \mathbb{N}$，low $_k(v)$ 可能不存在（这意味着 $\text{low}_{k'}(v)$ 不存在，适用于

任意 $k' > k$）。然而，如果 $low_k(v)$ 存在，那么对于任意 $k' < k$，$low_{k'}(v)$ 也存在，并且有$low_1(v) < low_2(v) < \ldots < low_k(v)$。注意，$low_k(v)$ 的存在意味着存在一条回边 $(x, low_k(v))$，其中 x 是 v 的后代。

命题 2.1。设 T 是简单图 G 的一棵深度优先搜索树，并假设 G 中每个顶点的邻接表都按照深度优先搜索编号递增排序。再假设，对于某个 $k \in \{0, \ldots, n-1\}$，我们已经计算出所有顶点相对于 T 的 low_1、……、low_k 点，并且对于每个 $v \neq r$，集合 $\{low_1(v)……、low_k(v)\}$ 都存储在一个递增有序数组中。那么，我们可以在 $O(n \log(k+1))$ 时间内计算出所有顶点的 low_{k+1} 点。[3]

证明。对于每个 $v \neq r$，令 $lowArray(v)$ 表示一个数组，其中按递增顺序存放 $\{low_1(v)、……、low_k(v)\}$，并额外包含一个值为 $null$ 的元素。现在，我们以自底向上的方式处理顶点（例如，按逆深度优先搜索顺序处理）。我们将确保，在开始处理某个顶点时，其子节点的 low_1、……、low_{k+1} 点都已正确计算（$*$）。

顶点 $v \neq r$ 的处理过程如下。首先，我们在 v的邻接表的前 $k+1$ 个条目中执行二分搜索，以找到大于 $low_k(v)$ 的最小顶点；如果存在，就将其插入 lowArray（v）的第 $k+1$个条目中。接下来处理 v的子节点。对于 v的每个子节点 c，如果 lowArray（v）的第 $k+1$个条目为 null，则我们在 lowArray（c）中执行二分搜索，以找到大于 $low_k(v)$ 且小于 v的最小顶点。如果存在，就将其插入 lowArray（v）的第 $k+1$ 个条目中。否则，如果 lowArray（v）的第 $k+1$个条目不为 null，则我们在 lowArray（c）中执行二分搜索，以找到大于 $low_k(v)$ 且小于 lowArray（v）第 $k+1$个条目的最小顶点。如果存在，就用 y替换 lowArray（v）第 $k+1$个条目中的顶点。注意，处理 v需要 $O((1+\text{nChilden}_v)\log(k+1))$ 的时间，其中 nChilden_v 是 v 的子节点数。因此，整个算法总共需要 $O(n\log(k+1))$ 的时间。

现在我们需要论证这一过程的正确性。假设对于顶点 v，$(*)$ 恰好在我们开始处理它时为真。（如果 v 是叶节点，则 $(*)$ 显然为真。）我们还假设 $low_k(v)$ 存在，因为否则 $low_{k+1}(v)$ 不存在，而我们就完成了。考虑 v 的邻接表的前 $k+1$ 个条目组成的线段 $y_1, \ldots, y_{k+1}$。然后注意到，$low_{k+1}(v) \leq y_{k+1}$（如果 y_{k+1} 为 null，我们令此不等式显然成立）。这是因为 $low_1(v)$ 至少和 y_1 一样低，因此 $low_2(v)$ 至少和 y_2 一样低，依此类推。因此，如果 $low_{k+1}(v)$ 存在于 v 的邻接表中，它就等于 $y_1, \ldots, y_{k+1}$ 中大于 $low_k(v)$ 的最低的那个。否则，在搜索 v 的邻接表之后，我们只能得到：lowArray (v) 的第 $k+1$ 个条目（如果它不为 null）包含一个大于 $low_{k+1}(v)$ 的顶点。现在，对于 v 的子节点 $i \in \{1, \ldots, k+1\}$，我们检查它们的 low 点。（根据 $(*)$，这些值已被正确计算，并存储在 lowArray 数组中。）首先，像之前一样，我们注意到，对于 v 的任意子节点 c，$low_{k+1}(v)$ 至少和 lowArray (c) 中的第 $k+1$ 个条目一样低。因此，如果存在一个 v 的子节点 c，使得 lowArray (c) 包含 $low_{k+1}(v)$，那么这正是 lowArray (c) 中大于 $low_k(v)$ 的最小顶点，我们会将其正确地插入 lowArray(v) 的第 $k+1$ 个条目中。

由此可知，当我们完成对 v 的处理时，$lowArray(v)$ 的第 $k+1$ 个条目要么是 $null$（据此可推知 $low_{k+1}(v)$ 不存在），要么包含一个大于 $low_k(v)$ 的顶点，但该顶点不大于 v 的邻接表中大于 $low_k(v)$ 的前 $k+1$ 个条目中的任何一个，也不大于其任一子节点中大于 $low_k(v)$ 的前 $k+1$ 个 low 点。因此，$low_{k+1}(v)$ 已在 $lowArray(v)$ 的第 $k+1$ 个条目中正确计算得出。 □

推论 2.2。对于任意 $k \in \{1, \ldots, n-1\}$，所有顶点的 low_1、……、low_k点都可以在 $O(m + kn \log k)$ 时间内计算出来。

[3]我们约定 $\log(1) = 1$，因此计算 low_1 点所需的时间为 $O(n$).

证明。 由命题 2.1 可立即得出：我们首先使用桶排序对所有顶点的邻接表排序，然后按此顺序计算所有顶点的 $low_1, \ldots, low_k$ 点。这需要 $O(m+n)+O(n \log 1+n \log 2+\ldots+n \log k) = O(m+kn \log k)$ 的时间。 □

3 顶点失效算法

3.1 初始化数据结构

我们需要以下 要素，以便能够处理至多 $d_\star$ 个失效顶点 s.

(i) 以顶点 r 为根的 G 的深度优先搜索树 T。必须计算所有顶点的 ND 和 深度 （相对于 T）。我们用 G 的深度优先搜索编号来标识 T 的顶点。

(ii) T 上的层祖先数据结构。

（iii）关于 T 的 G 的回边集合上的二维范围空性数据结构。

（iv）对于每个 $i \in \{1, \ldots, d_\star\}$，所有顶点的 low_i 点。

（v）对于每个 $i \in \{1, \ldots, d_\star\}$，一棵以 r 为根的 T 的深度优先搜索树（DFS-tree） T_i，其中顶点的邻接表由其按相对于 low_i 点的递增顺序排序的子节点列表给出。

（vi）对于每个 $i \in \{1, \ldots, d_\star\}$，关于 T_i 的 G 的回边集合上的二维范围空性数据结构。

（i）中的 深度 值指的是 T 中顶点的深度。对于每个顶点 v，其定义为树路径 $T[r, v]$ 的大小。（因此，例如，深度$(r) = 1$。）在深度优先搜索过程中计算 深度 值需要额外的 $O(n)$ 时间。

（ii）中的层祖先数据结构用于回答如下形式的查询`QueryLA`（v, δ） $\equiv$：“返回深度为 δ 的 v 的祖先”。我们利用这些查询来找出那些是其他顶点祖先的顶点的子节点。（也就是说，给定 u 是 v 的后代，我们想知道哪个 v 的子节点是 u 的祖先。）对于我们的用途，使用 [1], 第3节中的方案就足够了。该方案以 O（$n \log n$）的时间预处理 T，从而能够以（最坏情况下） O（1）的时间回答层级祖先查询。

（iii）中的二维范围空性数据结构用于回答如下形式的查询`2D range`（$[X_1, X_2] \times [Y_1, Y_2]$） $\equiv$：“是否存在一条满足 $x \in [X_1, X_2]$ 和 $y \in [Y_1, Y_2]$ 的回边（x, y）？”[4]我们可以使用该数据结构的标准实现，其空间和预处理时间复杂度均为 O（$m \log n$），并且可以在（最坏情况下） O（log n）时间内回答查询（例如，参见 [3] 中的第5.6节）。这里的 m 因子不可避免，因为回边数量可能达到 $m - n + 1$。不过，我们注意到，如果采用更复杂的方案（例如 [2]），就可以降低空间和查询时间中的 $\log n$ 因子。

所有顶点的 low_1、...、$low_{d_\star}$ 点都可以在 $O(m + d_\star n \log d_\star) = O(m + d_\star n \log n)$ 时间内计算出来（推论 2.2）。显然，我们需要 $O(d_\star n)$ 空间来存储它们。

对于（v），我们只需从 r 出发，在 T 上执行 $d_\star$ 次深度优先搜索，每次都使用 T 的子节点列表的不同排列作为邻接表。总共需要 $O(d_\star n)$ 时间，但我们不必实际存储这些树。（事实上，它们的父指针都相同。）这里实际需要的是对每个 $i \in \{1, \ldots, d_\star\}$，第 i 次深度优先搜索遍历的深度优先搜索编号，我们将其记为 DFS_i。我们会存储这些深度优先搜索编号，因此还需要 $O(d_\star n)$ 的额外空间。执行所有这些深度优先搜索的用处将在之后变得清楚

[4]二维范围查询的输入仅为查询矩形的端点 X_1、X_2、Y_1、Y_2；我们在端点周围使用括号，并使用符号 $\times$，仅为便于阅读。

在第 3.4 节中。目前，我们只需指出，对于每个 $i \in \{1, \ldots, d_\star\}$， T_i 中的祖先关系与 T 中的相同。因此，相对于 T_i ，所有顶点的 low_1、.....、$low_{d_\star}$ 点，与相对于 T 的结果相同。

(vi) 中的二维范围空性数据结构用于回答以下形式的查询： `2D range i`$([X_1, X_2] \times [Y_1, Y_2])$ $\equiv$“ 是否存在一条回边（ x, y ），满足 $x \in [X_1, X_2]$ 和 $y \in [Y_1, Y_2]$? ”，其中查询矩形的端点指的是深度优先搜索 $_i$ 编号，适用于$i \in \{1, \ldots, d_\star\}$。由于祖先关系对于 T_i 和 T相同，因此查询 `2D range`$([X_1, X_2] \times [Y_1, Y_2])$和 `2D range i`$([X_1, X_2]_i \times [Y_1, Y_2]_i)$等价，其中括号下方的 i下标表示我们已将端点转换为深度优先搜索 $_i$ 编号。

关于深度优先搜索树 $T_1, \ldots, T_{d_\star}$ 的二维范围空性数据结构的构建总共需要 $O(d_\star m \log\ n)$ 时间。为了存储这些数据结构，我们需要 $O(d_\star m \log\ n)$ 空间。因此，二维范围空性数据结构的构建和存储主导了整体的时空复杂度。

很容易看出，从 (i) 到 (vi) 的数据结构列表相对于 $d_\star$ 是灵活的。因此，如果 $d_\star$ 增加 1，我们还需要计算所有顶点的 $\text{low}_{d_\star+1}$ 点、 $T_{d_\star+1}$ 深度优先搜索树，以及相应的二维范围空性数据结构。计算 $\text{low}_{d_\star+1}$ 点需要 $O(n \log(d_\star + 1)) = O(n \log\ n)$ 时间，并额外占用 $O(n)$ 空间；这里假设我们已将 G 的邻接表按递增顺序排序，并且已将每个顶点的 $\text{low}1, \ldots,$ $\text{low}_{d_\star}$ 点存储在一个递增排序的数组中（见命题 2.1）。

推论 3.1. 假设我们已经针对某个 $d_\star$ 初始化了数据结构，现在希望获得适用于 $d_\star + k$ 的数据结构。那么，我们可以在 $O(km \log\ n)$ 时间内完成此操作，并额外使用 $O(km \log\ n)$ 空间。

如果 $d_\star$减少 k，那么我们只需丢弃低$_{d_\star-k+1}$、.....、低 $_{d_\star}$个点、$T_{d_\star-k+1}$、.....、 $T_{d_\star}$棵深度优先搜索树及相应的二维范围空性数据结构。这样将释放 $O(km$对数 $n)$的空间。

3.2 一般思想

设 F为失效顶点的集合。那么， $T \setminus F$可能由多个连通分量组成，而这些连通分量都是 T的子树。我们需要区分 $T \setminus F$的两类连通分量。设 C为 $T \setminus F$的一个连通分量。如果 F中没有顶点是 C的后代，那么C称为 $T \setminus F$的一棵悬挂子树。否则， C称为 $T \setminus F$的一个内部连通分量。（图1对此进行了说明。）注意，尽管$T \setminus F$的连通分量数目可能多达 $n - 1$（即使 $|F| = 1$）， $T \setminus F$的内部连通分量数目至多为 $|F|$。这是一个重要观察，它使我们能够将 $G \setminus F$的连通性归约为内部连通分量的连通性。

更准确地说，我们现在已经可以对如何回答顶点对之间的连通性查询给出一个高层次的策略描述。设 x， y是 $G \setminus F$中的两个顶点。首先，假设 x 属于一个内部连通分量 C_1 ，且 y 属于一个内部连通分量 C_2。那么，只需知道 C_1 和 C_2 在 $G \setminus F$中是否连通即可。否则，如果 x 或 y 中的任一个位于悬挂子树 C中，那么我们可以用与 C 在 $G \setminus F$中连通的任意内部连通分量替代 C 。如果不存在这样的内部连通分量，那么，当且仅当 x 和 y 位于同一个悬挂子树中时，它们才在 $G \setminus F$ 中连通。

因此，在从 G 中删除 F 后，只需做好准备，以便能够高效地回答以下问题：

(1) 给定一个顶点 x，确定包含 x的 $T \setminus F$ 的连通分量 .

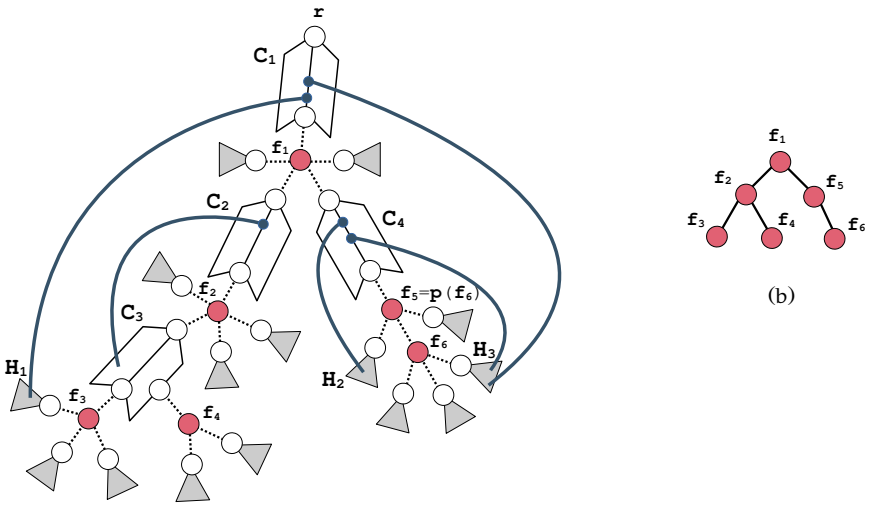

图1： (a) DFS树 T上的一组失效顶点 $F = \{f_1, \ldots, f_6\}$，以及(b)对应的F-森林，其中展示了失效顶点之间的父级$_F$ 关系。注意， $T \setminus F$ 被分成了多个连通分量，但只有四个内部连通分量： C_1、 C_2、 C_3 和 C_4。 $T \setminus F$ 的悬挂子树以灰色显示（例如 H_1、 H_2 和 H_3）。内部连通分量C_2 和 C_3 通过一条直接连接它们的回边，在 $G \setminus F$ 中保持连通。 C_1 和C_4 通过 f_6的悬挂子树 H_3 保持连通。我们有 $\partial(C_1) = \{f_1\}$、 $\partial(C_2) = \{f_2\}$、$\partial(C_3) = \{f_3, f_4\}$ 和 $\partial(C_4) = \{f_5\}$。注意， f_6 是唯一一个不是内部连通分量边界顶点的失效顶点，并且它的父级$_F(f_6) = p(f_6)$。

(2) 给定 $T \setminus F$ 的两个内部连通分量 C_1 和 C_2，判断 C_1 和 C_2 在 $G \setminus F$ 中是否连通。
(3) 给定 $T \setminus F$ 的一个悬挂子树 C，找到 $T \setminus F$ 的一个内部连通分量，使其在 $G \setminus F$ 中与 C 连通，或者报告不存在这样的内部连通分量。

事实上，最困难的任务，也是唯一需要在预处理阶段（更新阶段期间）处理的任务，是（2）。我们将在第3.5节中说明如何在回答查询的过程中执行（1）和（3）。对于（2），第3.4节给出了一个高效的解决方案。

基本思路是，由于最多只有 $d = |F|$ 个 $T \setminus F$内部连通分量，因此我们可以构造一个包含 $O(d)$个节点的图，用来表示 $T \setminus F$的内部连通分量，并捕捉它们之间的连通关系 $G \setminus F$（见引理3.8）。具体而言，我们在这些内部连通分量（的代表节点）之间引入若干人工边。在下一小节中，我们将陈述一些关于内部连通分量结构及其在 $G \setminus F$中的连通关系的引理。

3.3 内部连通分量的结构

我们将 $T \setminus F$ 的连通分量的根（视为 T 的有根子树）作为它们的代表。现在引入一些术语和记号。若 C 是 $T \setminus F$ 的一个连通分量，我们将其根记为 r_C。若 C 是 $T \setminus F$ 的一个悬挂子树，那么 $p(r_C) = f$ 是一个失效顶点，并且我们称 C 为 f 的一个悬挂子树。若 C，C' 是 $T \setminus F$ 的两个不同连通分量，且 $r_{C'}$ 是 r_C 的祖先，则称 C' 是 C 的祖先。此外，若 v 是一个不在 C 中的顶点，且 v 是 r_C 的祖先（分别为后代），则称 v 是 C 的祖先（分别为后代）。若 C 是一个

$T \setminus F$ 的内部连通分量，且 f 是一个失效顶点，使得 $p(f) \in C$，那么我们称 f 是 C 的边界顶点。C 的所有边界顶点组成的集合记为 $\partial(C)$。注意，任意顶点 $b \in \partial(C)$ 都具有这样的性质：树路径$T[p(b)\ r_C]$上没有失效顶点。反过来，若失效顶点 b 满足树路径 $T[p(b)\ r_C]$ 上没有失效顶点，则 b 是 C 的边界顶点。因此，若 $b_1, \ldots, b_k$ 是 C 的所有边界顶点组成的集合，那么 $C = T(r_C) \setminus (T(b_1) \cup \ldots \cup T(b_k))$。

以下引理是内部组件所满足的一组性质。

引理 3.2. 设 C 是 $T \setminus F$ 的内部连通分量。那么：

(1) 要么是 $r_C = r$，要么是 $p(r_C) \in F$。*(2)* 对于每个作为 C 后代的顶点 v，都存在唯一一个属于 C 且为 v 祖先的边界顶点。*(3)* 设 $f_1, \ldots, f_k$ 为 C 的边界顶点，并按递增顺序排列。那么 C 是以下连续顶点子集的并集：$[r_C, f_1 - 1]$，$[f_1 + \mathit{ND}(f_1)\ f_2 - 1]$，$\ldots$，$[f_{k-1} + \mathit{ND}(f_{k-1})\ f_k - 1]$，$[f_k + \mathit{ND}(f_k)\ r_C + \mathit{ND}(r_C) - 1]$。（注意，其中某些集合可能为空。）

证明。(1) 如果 $r_C \neq r$，则定义了 $p(r_C)$。由于 r_C 是 $T \setminus F$ 的一个连通分量的根，因此我们有 $r_C \notin F$。如果 $p(r_C) \notin F$，那么 r_C 就能通过父边 $(r_C, p(r_C))$ 在 $T \setminus F$ 中与 $p(r_C)$ 连通，这与 r_C 是 $T \setminus F$ 的一个连通分量的根这一事实矛盾。因此，$p(r_C) \in F$。

(2) 由于 v 是 C的后代，因此 $v \notin C$ 和 v 是 r_C的后代。由于v是 r_C的后代，因此树路径 $T[v, r_C]$上的所有顶点都是 v的祖先。（注意，只有 $T[v, r_C]$上的顶点才能既是 v的祖先，又是C的边界顶点，因为 v 的所有其他祖先都低于 r_C。）由于 $v \notin C$，$T[v, r_C]$上必定至少存在一个失效顶点。取其中最低的失效顶点 b。于是，树路径 $T[p(b)\ r_C]$ 上的顶点都不是失效顶点，因此 $p(b)$与 r_C在$T \setminus F$中连通，故 b 是 C的边界顶点。

现在，为了导出矛盾，假设 $T[v, r_C]$上还有另一个顶点 b'是 C的边界顶点。由于 b 是具有此性质的最低顶点，因此 b' 是 b的真后代。由于 $b' \in \partial(C)$，树路径 $T[p(b')\ r_C]$上不可能存在失效顶点，这与 $b \in T[p(b')\ r_C]$矛盾。因此，b是 $\partial(C)$中唯一一个作为 v祖先的顶点。

(3) 以 r_C 为根的子树由 $[r_C, r_C + \mathrm{ND}(r_C) - 1]$ 中的顶点组成。由于 $f_1, \ldots, f_k$ 是 C 的边界顶点，因此我们有 $C = T(r_C) \setminus (T(f_1) \cup \ldots \cup T(f_k))$。因此，$C = [r_C, r_C + \mathrm{ND}(r_C) - 1] \setminus ([f_1, f_1 + \mathrm{ND}(f_1) - 1] \cup \ldots \cup [f_k, f_k + \mathrm{ND}(f_k) - 1])$。由此，由于 $f_1, \ldots, f_k$ 按递增顺序排列，我们有 $C = [r_C, f_1 - 1] \cup [f_1 + \mathrm{ND}(f_1)\ f_2 - 1] \cup \ldots \cup [f_{k-1} + \mathrm{ND}(f_{k-1})\ f_k - 1] \cup [f_k + \mathrm{ND}(f_k)\ r_C + \mathrm{ND}(r_C) - 1]$。□

我们使用一片森林来表示失效顶点之间的祖先关系，并将其称为失效顶点森林（简称*F*-森林）。F-森林由以下两个元素组成。首先，对于每个失效顶点 f，都有一个父节点$_F(f)$指针，指向 f（在 T 中）中距离最近且同样是失效顶点的祖先。如果 f 没有失效顶点祖先，那么我们令父节点$_F(f) = \perp$。其次，每个失效顶点 f 都有一个指针，指向其在 F-森林中的子节点列表。

F-森林可以在 $O(d^2)$时间内轻松构建：我们只需为每个失效顶点 f找到其真祖先中编号最大的失效顶点 f'；然后设置$parent_F(f) = f'$，并将 f追加到F-森林中 f'的子节点列表中。

下一个引理说明了如何在常数时间内检查失效顶点是否属于内部连通分量的边界，以及如何获取该连通分量的根。

引理 3.3。 当且仅当 $parent_F(f) \neq p(f)$ 时，失效顶点 f 是某个内部连通分量的边界顶点。现在设 f是内部连通分量 C的边界顶点。那么，如果 $parent_F(f)$存在，则 C的根是 $parent_F(f)$的一个子节点，且该子节点是f的祖先。否则， C 的根就是 r。

证明。设 C 是一个内部连通分量，且满足 $f \in \partial(C)$。那么，树路径 $T[p(f)\ r_C]$上不存在失效顶点。特别地， $p(f) \neq \text{parent}_F(f)$。反过来，假设 $\text{parent}_F(f) \neq p(f)$。（我们可以排除 $f = r$这种情况，因为此时 $\text{parent}_F(f)$ $p(f)$ 这些表达式均未定义。）如果 $\text{parent}_F(f)$未定义，那么树路径 $T[p(f)\ r]$（即 f的祖先所在的路径）上不存在失效顶点，因此 f是根为 r的内部连通分量的边界顶点。否则，如果 $\text{parent}_F(f)$已定义，则$p(f)$不可能是失效顶点（否则会有 $\text{parent}_F(f) = p(f)$，因为 $\text{parent}_F(f)$是 f的最近的真祖先，且该祖先是失效顶点）。因此， $p(f)$属于 $T \setminus F$的一个连通分量，而 f 是该连通分量的边界顶点。

现在令 f 为某个内部连通分量 C 的边界顶点。这意味着树路径 $T[p(f)\ r_C]$ 上没有失效顶点。如果父节点 $F(f)$ 存在，那么它必定是 r_C 的真祖先。因此， $r_C \neq r$，由此根据引理 3.2(1)，我们有 $p(r_C)$ 是一个失效顶点。由于 $\text{parent}_F(f)$ 是 f 的最近失效祖先，因此我们有 $\text{parent}_F(f) = p(r_C)$，所以 r_C 是 $\text{parent}_F(f)$ 的子节点，且是 f 的祖先。否则，如果 $\text{parent}_F(f)$ 不存在，这意味着树路径 $T[p(f)\ r]$ 上没有失效顶点。因此， f 是根为 r 的内部连通分量的边界顶点。

□

因此，根据引理 3.3，如果 f 是内部连通分量 C 的边界顶点，且该分量的根为$r_C \neq r$，我们可以通过一次层级祖先查询在常数时间内检索 r_C：即，查询 f（在 T 中）深度与 父节点$F(f)+1$相同的祖先。此后我们可以不再特别说明，直接使用这一事实。

以下引理表明，有两类边决定了 $G \setminus F$ 的连通分量之间的连通关系 $T \setminus F$。

引理 3.4. 设 e 是 $G \setminus F$ 中一条边，其端点位于 $T \setminus F$ 的不同连通分量中。那么 e 是一条回边，并且要么 (i) e 的两个端点都位于内部连通分量中，要么 (ii) e 的一个端点位于悬挂子树 H 中，另一个端点位于 C 内部连通分量中，而 C 是 H 的祖先。

证明。设 $e = (x, y)$，设 C 是包含 x 的 $T \setminus F$ 的连通分量，并设 C' 是包含 y 的 $T \setminus F$ 的连通分量。我们知道 e 不可能是树边，否则 x 和 y 将在 $T \setminus F$ 中连通。因此， e 是一条回边。由于 x、y 是一条回边的端点，它们之间存在祖先与后代关系。不失一般性，我们可以假设 $r_C > r_{C'}$。我们将证明这意味着 x 是 y 的后代。现在，为了导出矛盾，假设 x 是 y 的祖先。由于 $y \in C'$，我们知道 y 是 $r_{C'}$ 的后代。由于 x 是 y 的祖先且不在 C' 中，我们知道 x 不在树路径 $T[y, r_{C'}]$ 上。因此， x 是 $r_{C'}$ 的真祖先，所以 $x < r_{C'}$。由于 $x \in C$，我们知道 $x \geq r_C$。因此，我们得到 $r_C \leq x < r_{C'}$，这与假设 $r_C > r_{C'}$ 矛盾。由此，我们证明了 x 是 y 的后代。现在，由于 y 不在 C 中，我们知道 y 不可能位于树路径 $T[x, r_C]$ 上。因此，由于 y 是 x 的祖先，它必定是 r_C 的真祖先。而且，由于 $y \in C'$，我们知道 y 是 $r_{C'}$ 的后代。因此，r_C 是 $r_{C'}$ 的后代。

因此，我们已经证明 C 是 C'的后代。这意味着 C' 不可能是 $T \setminus F$的悬挂子树。为说明这一点，假设情况相反。由于 $r_{C'}$是 r_C的真祖先，我们有

即 $r_{C'}$ 是 $p(r_C)$ 的祖先。（$p(r_C)$ 之所以有定义，正是因为 r_C 存在一个真祖先，因此 $r_C \neq r_\circ$。）注意，$p(r_C)$ 是一个失效顶点（否则，r_C 将通过父边 $(r_C, p(r_C))$ 与 $p(r_C)$ 连通，这与 r_C 是 $T \setminus F$ 的一个连通分量的根这一事实相矛盾）。然而，这意味着 r'_C 是某个失效顶点的祖先，这与我们关于 C' 是 $T \setminus F$ 的悬挂子树的假设相矛盾。由此我们得出结论：在 C 和 C' 中，只有 C 能够是 $T \setminus F$ 的悬挂子树。 □

推论 3.5. 设 C、C' 是 $G \setminus F$ 的两个不同连通分量，它们通过 e 中的一条边相连。不失一般性，假设 $r_{C'} < r_C$。那么 C' 是 C 的祖先。

证明。引理 3.4 意味着 e 是一条回边。令 $e = (x, y)$，并且不失一般性地假设 x 是 y 的后代。我们知道，要么 $x \in C$ 和 $y \in C'$，要么反之。为导出矛盾，假设 $x \in C'$（因此 $y \in C$）。这意味着 x 是 $r_{C'}$ 的后代。因此，x 是 $r_{C'}$ 和 y 的共同后代。这意味着 $r_{C'}$ 和 y 互为祖先与后代。我们知道，y 不可能是 $r_{C'}$ 的后代，因为这将意味着 $y \in T[r_{C'}, x]$（但 y 位于 C' 之外）。因此，我们有 y 是 $r_{C'}$ 的真祖先，故 $y < r_{C'}$。由于 $r_{C'} < r_C$，这意味着 $y < r_C$，因此 y 不可能是 r_C 的后代——这与 $y \in C$ 这一事实矛盾。

因此，我们已证明 $x \in C$ 和 $y \in C'$。 $x \in C$ 意味着 x 是 r_C的后代。因此，x 是 r_C 和 y的共同后代。这意味着 r_C 和 y 之间具有祖先与后代的关系。我们知道，y 不可能是 r_C的后代，因为这将意味着 $y \in T[r_C, x]$（但 y位于 C之外）。因此，r_C是 y的后代。此外，$y \in C'$意味着 y 是 $r_{C'}$的后代。因此，我们得出结论：r_C 是 $r_{C'}$的后代。 □

下面的引理给出了一个在算法上有用的判据，用于判断 $T \setminus F$ 的某个连通分量——悬挂子树或内部连通分量——是否通过一条回边与 $T \setminus F$ 的某个内部连通分量相连。

引理 3.6. 设 C、C' 是 $T \setminus F$ 的两个连通分量，其中 C' 是 C的祖先且属于内部连通分量，并设 b 是 C的祖先且属于 C' 的边界顶点。那么，当且仅当存在一条从 C 到 C' 的回边时，存在一条回边，其下端点位于 $[r_{C'}, p(b)]$中的 C 。

证明。首先，我们来说明 b的存在性。由于 C' 是 C的祖先，因此 $r_{C'}$ 是 r_C的祖先。因此，引理 3.2(2) 表明，存在唯一一个 b的边界顶点 C'，它是 r_C的祖先。因此，b 是 C的祖先。

(⇒) 令 $e = (x, y)$ 是一条从 C 到 C'的回边，并且不失一般性地假设 x 位于 C中。由于 e 是一条回边，因此 x 是 y的后代，或者反过来。为导出矛盾，假设 y 是 x的后代。由于 $x \in C$，我们有 $x \geq r_C$。由于y是 x的后代，因此 $y > x$。于是，$y > r_C$。由于(x, y) 是一条从 C到C' 的回边，并且 $x \in C$，因此 $y \in C'$。这意味着树路径 $T[y, x]$上必定有一个失效顶点。（否则，y会与 x连通，因此 C'会与 C连通，这显然不可能。）令 f为树路径 $T[y, x]$ 上的最大失效顶点。包含 y 的 $T \setminus F$ 的连通分量以 f 的一个子节点为根。但这个根是$r_{C'}$，因此我们有 $r_{C'} > f > x \geq r_C$，这与 C' 是 C的祖先这一假设矛盾。由此我们证明了 x 是 y 的后代。由于 y 是x的祖先且不位于 C中，因此 y不位于树路径 $T[x, r_C]$上，故它必定是 r_C 的真祖先。因此，由于 $y \in C'$，我们有 y 位于树路径$T[p(b)\ r_{C'}]$上。这意味着 $y \in [r_{C'}, p(b)]$。

(⇐) 令 $e = (x, y)$ 为一条来自 C 的回边，其下端点位于 $[r_{C'}, p(b)]$ 中。不失一般性，我们可以假设 $x \in C$。因此，我们有 $y \in [r_{C'}, p(b)]$, ，并且 y 是 x 的祖先。由于$x \in C$，我们有 x 是 r_C 的后代。由于 b 是 C 的祖先，我们有 b 是一个

r_C 的祖先。因此， x 是 b 的后代，因而也是 $p(b)$ 的后代。这意味着树路径 $T[p(b)\ r_{C'}]$ 上的顶点都是 x 的祖先。因此，由于 y 是 x 的祖先，且$y \in [r_{C'}, p(b)],$，我们有 $y \in T[p(b)\ r_{C'}]$。由于 b 是 C' 的边界顶点，我们有树路径 $T[p(b)\ r_{C'}]$ 上的所有顶点都位于 C' 中。特别地，我们有 $y \in C'$。

□

定义 3.7。令 $\mathcal{R}$ 为一个多重图，其中 $V(\mathcal{R})$ 是 $T \setminus F$ 的内部连通分量的根集合，并且 $E(\mathcal{R})$ 满足以下三个性质：

(1) 对于连接两个内部连通分量 C 和 C'的每条回边， $\mathcal{R}$中存在一条边 $(r_C, r_{C'})$。*(2)* 设 H是失效顶点 f的一个悬挂子树，并设 $C_1, \ldots, C_k$是通过一条回边与 H相连的内部连通分量。（根据引理 *3.4*， $C_1, \ldots, C_k$都是 H的祖先。）不失一般性地，设 C_k是所有 $C_1, \ldots, C_{k-1}$ 的祖先。那么 $\mathcal{R}$ 包含边 (r_{C_1}, r_{C_k}) 、 $(r_{C_2}, r_{C_k}) \ldots, (r_{C_{k-1}}, r_{C_k})$。*(3)* $\mathcal{R}$ 的每条边要么由 *(1)* 或 *(2)* 给出，要么是形如 $(r_C, r_{C'})$的边，其中 C, C' 是 $G \setminus F$中相连的两个内部连通分量。

那么， $\mathcal{R}$ 被称为 $T \setminus F$的内部连通分量的连通性图 。(1)和(2)中的边分别称为类型 *1* 和类型 *2*。

下面的引理表明，该图捕捉了 $G \setminus F$中 $T \setminus F$的内部连通分量之间的连通关系。

引理 3.8。设 $\mathcal{R}$ 是 $T \setminus F$的内部连通分量的连通性图。那么， $T \setminus F$中的两个内部连通分量 C、C'在 $G \setminus F$中连通，当且仅当 r_C、$r_{C'}$在 $\mathcal{R}$中连通。

证明。（⇒）设 C、C'是 $T \setminus F$中在 $G \setminus F$里连通的两个内部连通分量。这意味着存在一列 $T \setminus F$的两两不同的连通分量 $C_1, \ldots, C_k$，以及一列回边 $e_1, \ldots, e_{k-1}$，使得： $C_1 = C$、$C_k = C'$，并且对每个 $i \in \{1, \ldots, k-1\}$， e_i连接 C_i和C_{i+1}。根据引理 3.4，对于每个 $i \in \{1, \ldots, k-1\}$，要么（1） C_i 和 C_{i+1} 是内部连通分量，且二者具有祖先与后代关系；要么（2） C_i、C_{i+1} 中一个是悬挂子树，另一个是作为它祖先的内部连通分量。

设 i 是 $\{1, \ldots, k-1\}$ 中的一个索引。如果 (1) 成立，那么 $\mathcal{R}$ 中存在一条 Type-1 边 $(r_{C_i}, r_{C_{i+1}})$。如果 (2) 成立，那么 C_i、C_{i+1} 中有一个是悬挂子树。假设 C_i 是一棵悬挂子树。由于有一条回边连接 C_i 和 C_{i+1}，我们可以考虑 $\tilde{}$ 中既是 C_i 的祖先、又通过一条回边与之相连的最低的内部连通分量 C。如果 $C_{i+1} = C$，那么此处无需推断任何结论。否则，我们有 $\mathcal{R}$ 包含 Type-2 边 $(r_{C_{i+1}}, r_{\tilde{C}})$。现在，由于 C_1 是一个内部连通分量，我们有 $C_i \neq C_1$，因此 C_{i-1} 有定义。根据引理 3.4，我们有 C_{i-1} 也是一个内部连通分量，$\tilde{}$ 通过一条回边与 C_i 相连。同样，如果 $C_{i-1} = C$，那么此处无需推断任何结论。否则，我们有 $\mathcal{R}$ 包含 Type-2 边 $(r_{C_{i-1}}, r_{\tilde{C}})$。因此，需要考虑三种可能性： $C_{i-1} = C$ 和 $C_{i+1} \neq C$，或 $C_{i-1} \neq C$ 和 $C_{i+1} = C$，或 $\tilde{}$ $\tilde{}$ $C_{i-1} \neq C$ 和 $C_{i+1} \neq C$。无论哪种情况，我们都可以看到， $r_{C_{i-1}}$ 在 $\mathcal{R}$ 中与 $r_{C_{i+1}}$ 相连——可以直接相连，也可以通过 $r_{\tilde{C}}$ 相连。类似地，假设 C_{i+1} 是一棵悬挂子树（而 C_i 是一个内部连通分量），则有 r_{C_i} 在 $\mathcal{R}$ 中与 $r_{C_{i+2}}$ 相连。

由此我们推断，如果 $C_{i(1)}, \ldots, C_{i(t)}$ 是由内部连通分量组成的 $C_1, \ldots, C_k$ 的子序列，那么 $r_{C_{i(1)}}, \ldots, r_{C_{i(t)}}$ 在 $\mathcal{R}$ 中连通。特别地，我们有 r_C 和 $r_{C'}$ 在 $\mathcal{R}$ 中连通。

（⇐）设 $e = (r_C, r_{C'})$ 是 $\mathcal{R}$ 的一条边。如果 e 是一条 Type-1 边，那么存在一条回边，在 $G \setminus F$ 中连接 C 和 C'。否则， $T \setminus F$ 中存在一个悬挂子树，通过回边与 $G \setminus F$ 中的 C 和 C' 均相连。无论哪种情况，我们都有 C, C' 在 $G \setminus F$ 中连通。由于这对 $\mathcal{R}$ 的任意一条边都成立，我们得出：若 $r_C, r_{C'}$ 是 $\mathcal{R}$ 中两个相连的顶点，则 C, C' 在 $G \setminus F$ 中连通。 □

3.4 处理更新：为 $T \setminus F$ 的内部连通分量构建连通性图

给定失效顶点集合 F，其中 $|F| = d \leq d_\star$，我们将说明如何使用 $O(d^4)$ 次二维范围空性查询，构建 $\mathcal{R}$，以表示 $T \setminus F$ 的内部连通分量的连通性图。回忆一下， $V(\mathcal{R})$ 是 $T \setminus F$ 的内部连通分量的根集合。

算法 1 展示了如何找出 $\mathcal{R}$ 的所有 Type-1 边。基本思路是针对每对内部连通分量执行二维范围空性查询，以确定是否存在连接它们的回边。具体而言，我们按如下方式处理。设 C 是 $T \setminus F$ 的一个内部连通分量。然后，只需检查 C 的每个祖先连通分量 C'，即可确定是否存在一条从 C 指向 C' 的回边（见推论 3.5）。设 $f_1, \ldots, f_k$ 为 C 的边界顶点，并按递增顺序排列。再设 f' 为 C' 的边界顶点，且它是 C 的祖先，并令 $I = [r_{C'}, p(f')]$。然后，我们在矩形 $[r_C, f_1 - 1] \times I, [f_1 + \text{ND}(f_1)\ f_2 - 1] \times I, \ldots, [f_k + \text{ND}(f_k)\ r_C + \text{ND}(r_C) - 1] \times I$ 上执行二维范围空性查询，以确定是否存在回边。当且仅当这些查询中至少有一个返回肯定结果时，我们才知道存在连接 C 和 C' 的回边（见引理 3.2(3) 和引理 3.6）。若确实如此，则将边 $(r_C, r_{C'})$ 添加到 $\mathcal{R}$。

注意，我们执行的二维范围空性查询总数为 $O(d^2)$，因为每个查询都对应一个三元组 (C, f, C')，其中 C，C' 是内部连通分量， C' 是 C的祖先，且 f 是 C的边界顶点，或者 r_C。如果 C_1， $\ldots$， C_k是 $T \setminus F$的所有内部连通分量，那么这些三元组的数量有界为 $(|\partial(C_1)| + 1) \cdot d + \ldots + (|\partial(C_k)| + 1) \cdot d = (|\partial(C_1)| + \ldots + |\partial(C_k)| + k) \cdot d \leq (d + k) \cdot d \leq (d + d) \cdot d = O(d^2)$。

命题 3.9。 算法1能够正确计算所有 *Type-1* 边，以便为 $T \setminus F$ 的内部连通分量构建连通性图。该算法的运行时间为 $O(d^2 \log n)$。

证明。 首先，我们需要提供一种方法，以高效遍历内部连通分量及其边界顶点的集合（第 1 和 2行），然后证明第 **while** 行的 5 循环足以访问所有作为 C祖先的内部连通分量。然后，正确性和 $O(d^2 \log n)$ 时间界由上述分析（正文中）得出。

每个内部连通分量 C $T \setminus F$由其根 r_C确定。根据引理3.2(1)，可知要么 $r_C = r$，要么 $p(r_C)$是一个失效顶点。如果 $r_C = r$，那么 C 没有祖先内部连通分量，因此可以忽略这种情况。于是设 $p(r_C) = f$ 是一个失效顶点。那么，根据引理3.3， C 的边界顶点由 f 在 F-森林中且是（在 T中） r_C的后代的子节点给出。

因此，我们可以按如下方式处理。首先，按递增顺序对 F-森林中每个失效顶点 f 的子节点排序。总共需要 $O(d \log d)$ 时间。然后，对于每个失效顶点f，我们按顺序遍历其子节点列表 L（在 F-森林中）。对于每个由同一子节点 c 的后代组成的极大线段 $S\ L$，我们知道，要么 c 是一个边界为 S 的内部连通分量的根，要么 $c \in F$。（对于遇到的每个 $f' \in L$，我们可以使用层级祖先查询，在常数时间内找到 f 在 T 中的子节点，该子节点是 f' 的祖先。）因此，第 1 行和第 2 行总共需要 $O(d \log d)$ 时间。

算法 1: 计算所有类型 1 边，以构建连通性图 $\mathcal{R}$，用于__日本 $T \setminus F$ 的内部连通分量

1 **对于每个** 内部连通分量 C 的 $T \setminus F$ **执行**
2 设 f_1、……、f_k 为 C 的边界顶点，按递增顺序排列。
3 // process every internal component C' that is an ancestor of C
4 设置 $f' \leftarrow p(r_C)$
5 **当** $f' \neq \perp$ **时执行**
6 **如果** $p(f') \neq$父节点$_F(f')$，**则**
7 令 C' 为 $T \setminus F$ 中满足 $f' \in \partial(C')$ 的内部连通分量
8 设置 $I \leftarrow [r_{C'}, p(f')]$
9 **如果** 以下查询中至少有一个为正：
10 2D_range($[r_C, f_1 - 1] \times I$)
11 2D-range($[f_1 + ND(f_1)\ f_2 - 1] \times I$)
12 ...
13 2D_range($[f_{k-1} + ND(f_{k-1})\ f_k - 1] \times I$)
14 2D-range($[f_k + ND(f_k)\ r_C + ND(r_C)\ - 1] \times I$) **那么**
15 将 Type-1 边 $(r_C, r_{C'})$ 添加到 $\mathcal{R}$
16 **end**
17 **end**
18 $f' \leftarrow$父节点$_F(f')$
19 **end**
20 **结束**

现在，设 C 是一个具有 $r_C \neq r$ 的内部连通分量。那么，$p(r_C)$ 是一个失效顶点。现在，设 C' 是 C 的一个真祖先内部连通分量。这意味着 $r_{C'}$ 是 r_C 的真祖先，因此 $r_{C'}$ 是 $f = p(r_C)$ 的祖先。然后，根据引理 3.2(2)，存在 C' 的一个边界顶点 b，它是 f（在 T 中）的祖先。由于在 T（在 f 中）中作为祖先的失效顶点集合，与 F-森林中 f 的祖先集合相同，因此第 5 行的 while 循环最终会到达 b。然后，我们可以使用引理 3.3 在常数时间内检索 C'（更准确地说：$r_{C'}$）。第 6 行的目的是应用引理 3.3 的判定准则，以检查 f' 是否是某个内部连通分量的边界顶点。 □

Type-2 边的构造并非那么直接。对于每个失效顶点 f，以及任意两个内部连通分量 C 和 C'，满足 C 是 f 的祖先且 C' 是 C的祖先，我们希望确定是否存在一个以 f为根的悬挂子树，其中有一条回边 e 的一个端点位于 C，还有一条回边 e' 的一个端点位于 C'。确定这一点的直接方法如下。设 b（分别为 b'）是 C（分别为 C'）中作为 f祖先的边界顶点。然后，对于每个根为 c的 f 悬挂子树，我们在矩形 $[c, c + ND(c)\ - 1] \times [r_C, p(b)]$ 以及$[c, c + ND(c)\ - 1] \times [r_{C'}, p(b')]$上执行二维范围空性查询。如果两个查询的结果均为肯定，则可知 C和 C'通过根为 c的悬挂子树在 $G \setminus F$中连通。

显然，这种方法通常效率不高，因为f 的悬挂子树数量可能非常接近 n。不过，它是我们更高效方法的基础。其思路是针对大批悬挂子树，一次执行大量此类查询。具体来说，我们对 f 的连续 悬挂子树执行查询（即它们的根是 f 的连续子节点），并且我们知道 C' 上的答案为肯定（也就是说，对于这些子树中的每一棵，都必然存在一条将其与 C'相连的回边）。要使这一思路奏效，我们必须适当重新排列所有顶点的子节点列表。（否则，悬挂

子树中与 C' 通过回边相连的 f，在 f 的子节点列表中可能并不连续。）实际上，我们维护多棵深度优先搜索树（具体而言： $d_\star$），以及多个二维范围空性数据结构，每种不同的子节点列表排列对应一个。

让我们详细阐述这个想法。设 H 是连接若干内部连通分量的 f 悬挂子树，并设 C' 是其中最低的一个（即它是所有其他分量的祖先）。那么，所有从 H 出发并终止于 C' 的祖先的回边，其下端点都是 C' 的祖先失效顶点。因此，由于失效顶点总数至多为 d，$\mathsf{low}_1(r_H), \ldots, \mathsf{low}_d(r_H)$ 中至少有一个属于 C'。换言之，对于某个 $i \in \{1, \ldots, d\}$，r_H 是 f 的一个子节点，且其 low_i 点属于 C'。现在，假设对于每个 $i \in \{1, \ldots, d_\star\}$，我们都存有一份 f 的子节点列表副本，该列表按 low_i 点递增顺序排列；将此列表记为 $L_i(f)$，并以支持按 low_i 点进行二分搜索的方式存储。那么，对于每个作为 f 祖先的内部连通分量 C，我们可以通过在 $L_i(f)$ 中查找 low_i 点位于 $[r_C, p(b)]$, 的最左和最右子节点，找到 $L_i(f)$ 中由 f 的子节点组成的线段 $S_i(C)$，这些子节点的 low_i 点属于 C；其中，b 是 C 的边界顶点，且该顶点是 f 的祖先。

现在令 $i \in \{1, \ldots, d\}$ 满足 $\mathsf{low}_i(r_H) \in C'$。于是我们有 $r_H \in S_i(C')$。此外，f 的每个子节点，只要它位于 $S_i(C')$ 中且是 f 的某个悬挂子树 H' 的根，就具有这样的性质：H' 也通过一条回边与 C' 连通。因此，我们希望能够像上文那样，对 $S_i(C')$ 的子集 S 执行二维范围空性查询；该子集由悬挂子树的根组成，用以确定 C' 在 $G \backslash F$中的连通性，以及它与所有作为 f 祖先且作为 C'后代的内部连通分量 C 之间的连通性。若能保证 S 由 f的大段连续子节点组成，我们就能高效地完成此操作。我们可以在预处理阶段满足这一条件：对于每个 $i \in \{1, \ldots, d_\star\}$，从 r 开始对 T执行深度优先搜索，其中每个顶点 v 的邻接表按$L_i(v)$给出。[5] 令 T_i 为得到的深度优先搜索树，DFS_i 为对应的深度优先搜索编号。然后，利用 T_i的深度优先搜索编号，我们初始化一个二维范围数据结构 i，用于在 $[1, n] \times [1, n]$的子矩形中回答相对于 T_i 的回边的二维范围空性查询。

现在让我们看看如何将所有内容组合起来。设 H 是 f 的一个悬挂子树，它连接两个内部连通分量 C_1 和 C_2；设 b_1 和 b_2 分别是 C_1 和 C_2 中属于 f 祖先的边界顶点。设 C' 是通过回边与 H 相连的最低的内部连通分量。那么存在一个 $i \in \{1, \ldots, d\}$，使得 $\mathsf{low}_i(r_H) \in C'$。设 S 是 $S_i(C')$ 中包含 r_H 且由悬挂子树的根组成的极大线段，设 L 是 S 的最小值，R 是 S 的最大值。[6] 于是，关于 $[L, R + \mathsf{ND}(R) - 1]_i \times [r_{C_1}, p(b_1)]_i$ 和 $[L, R + \mathsf{ND}(R) - 1]_i \times [r_{C_2}, p(b_2)]_i$、二维范围为 i 的二维范围空性查询结果均为肯定，因此我们将在 $\mathcal{R}$ 中添加边 $(r_{C_1}, r_{C'})$ 和 $(r_{C_2}, r_{C'})$。这样便会在 $\mathcal{R}$ 中维护如下信息：C'、C_1 和 C_2 与 f 的同一个悬挂子树相连。

用于构造足够多 Type-2 边以生成 $\mathcal{R}$ 的内部连通分量连通性图的算法 $T \setminus F$ 见算法2。正确性证明和时间复杂度分析见命题3.10。

命题 3.10。算法2 计算出足够多的 *Type-2* 边，以构造 $\mathcal{R}$ 的内部连通分量连通性图 $T \setminus F$（假设 $\mathcal{R}$ 包含所有 *Type-1* 边）。该算法的运行时间为 $O(d^4 \log n)$。

证明。 根据定义，只需证明以下内容：对于每个失效顶点 f，以及 f 的每个悬挂子树 H，令 C' 为通过回边与 H 相连的最低的内部连通分量；那么，对于每个通过回边与 H 相连的内部连通分量 $C \neq C'$，都会向 $\mathcal{R}$ 添加一条边 $(r_C, r_{C'})$。反过来，这些也都是全部内容。

[5]也就是说，v 的邻接表中的顶点必须按照与 $L_i(v)$相同的顺序出现。[6]注意，由于 T_i的构造，我们有：$DFS_i(L)$ 和 $DFS_i(R)$ 分别也是 $DFS_i(S)$ 的最小值和最大值。

算法 2： 计算足够多的 Type-2 边，以构造 $T \setminus F$ 的内部连通分量的连通性图

对每个失效顶点 f **执行**
  // process all pairs of internal components that are ancestors of f
  设置 $f' \leftarrow$ 的父节点$_F(f)$
  当 $f' \neq \perp$ **时执行**
    令 C' 为内部连通分量，其中 $f' \in \partial(C')$
    // skip the following if C' does not exist, and go immediately to Line 26
    对每个 $i \in \{1, \ldots, d\}$ **执行**
      令 $\mathcal{S}_i$ 为 $L_i(f)$ 中所有由悬挂子树的根组成的极大线段的集合 其 最低$_i$ 点位于 C' 中
    end
    // process all internal components C that are ancestors of f and descendants of C'
    设置 $f'' \leftarrow f$
    当 $f'' \neq f'$ **时执行**
      令 C 为具有 $f'' \in \partial(C)$ 的内部连通分量
      // skip the following if C does not exist, and go immediately to Line 24
      // check if C is connected with C' through at least one hanging subtree of f
      对每个 $i \in \{1, \ldots, d\}$ **执行**
        对每个 $S \in \mathcal{S}_i$ **执行**
          令 $L \leftarrow$ 最小值(S) 和 $R \leftarrow$ 最大值(S)
          如果 2D range i$([L, R + ND(R) - 1]_i \times [r_C, p(f'')]_i) =$ 真 **则**
            将 Type-2 边（r_C，$r_{C'}$）添加到 $\mathcal{R}$
          end
        end
      end
      $f'' \leftarrow$父节点$_F$（f''）
    end
    $f' \leftarrow$父节点$_F$（f'）
  end
结束

添加到 $\mathcal{R}$中的 Type-2 边，并且通过算法 r_C，$r_{C'}$添加到 $\mathcal{R}$中的任何其他边都具有这样的性质：算法 2 保证 C 和 C' 通过回边与同一个悬挂子树相连。

因此，设 f 是一个失效顶点，H 是 f 的一个悬挂子树，C' 是通过回边与 H 相连的最低内部连通分量。假设 $f \notin \partial(C')$。（否则，不存在既是 H 的祖先又是 C' 的后代的内部连通分量 C，因此 H 不会产生任何 Type-2 边。）设 $C \neq C'$ 是通过回边与 H 相连的一个内部连通分量。由于 C' 是通过回边与 H 相连的最低内部连通分量，根据上文（正文）中的分析，我们有一个 $i \in \{1, \ldots, d\}$，使得 最低的 $_i(r_H) \in C'$。因此，我们可以考虑 $L_i(f)$ 中包含 r_H 的极大线段 S；该线段由悬挂子树的根组成，且其 最低的 $_i$ 点位于 C' 中。设 L 和 R 分别为 S 的最小值和最大值。根据构造，

T_i，我们有 S 按照 深度优先搜索$_i$ 编号以递增顺序排列。因此，区间$[L, R + ND(R) - 1]_i$ 由 T_i 中 S 所含顶点的后代组成。由于 T_i 中的顶点具有与 T 中相同的祖先关系，因此集合 深度优先搜索$_i(S)$ 由 T_i 中 f 的子节点组成，这些子节点是悬挂子树的根，且其 最低的$_i$ 位于 C' 中。

现在令 f''为 C中作为 f祖先的边界顶点。那么我们也有，f''是 C中作为 r_H祖先的边界顶点（因为 $f = p(r_H)$）。因此，引理 3.6 表明，存在一条从 H到 $[r_C, p(f'')]$的回边。所以，相对于 T_i，也存在一条从深度优先搜索 $_i(H)$ 到 [深度优先搜索 $_i(r_C)$ 的回边，深度优先搜索 $_i(p(f''))$]。这意味着第 19 行的二维范围查询为真，因此会将类型 2 边 $(r_C, r_{C'})$ 正确地添加到 $\mathcal{R}$。不难看出，反过来也成立：每当第 19 行的二维范围查询为真，我们就可以确定，f 中存在一个悬挂子树，它通过一条回边同时与 C' 和 C相连。

下面分析算法 2 的运行时间。首先，我们将给出一种实现第 8 行的方法，即如何针对每个作为 f 祖先的内部连通分量 C'，以及每个 $i \in \{1, \ldots, d\}$，找出集合 $\mathcal{S}_i$：该集合由 $L_i(f)$ 的极大线段组成，这些线段由悬挂子树的根构成，且这些根的低$_i$ 点位于 C' 中。有许多种方法可以实现这一点，但为简单起见，我们将给出一种相对直接的方法，其总成本为$O(d^3 + d^2 \log n)$。其基本思想是在第 1 行 for 循环开始时，收集作为失效顶点祖先的 f 的子节点。为此，我们收集在 F-森林中作为 f 子节点的失效顶点 f_1、……、f_k，然后对每个 f_i 执行一次层级祖先查询（在 T 中），以找到作为 f_i 祖先的 f 的子节点 c_i。随后，我们保留集合 $\{c_1, \ldots, c_k\}$ 的 d 个副本，即 $\mathcal{C}_1$、……、$\mathcal{C}_d$。（注意，对于满足 $i \neq j$ 的 i、$j \in \{1, \ldots, k\}$，某些 c_i、c_j 可能会重合。我们忽略这些重复项。）对于每个 $i \in \{1, \ldots, d\}$，我们令 $\mathcal{C}_i$ 按照低$_i$ 点的递增顺序排序。现在，在第 7 行 for 循环开始时，我们可以使用二分搜索，在 $O(n)$ 时间内找到 $L_i(f)$ 中的线段 S（的端点）；该线段由所有低$_i$ 点位于 C' 中的 f 的子节点组成。令 L 和 R 分别为 S 的最小值和最大值。然后遍历列表 $\mathcal{C}_i$，并对遇到的每个 $c \in \mathcal{C}_i$ 检查 c 是否属于 S。（这只需检查 $L \leq c \leq R$ 是否成立即可。）如果成立，则收集可能为空的子线段 $[L, c-1]$（即其端点对），并从 S 中移除 $[L, c]$（即令 $L \leftarrow c + 1$）。我们在遍历 $\mathcal{C}_i$ 的过程中重复此过程，直到到达其末尾，最后收集 S 中可能为空的剩余部分（即其端点对）。我们收集到的非空子线段的集合正是$\mathcal{S}_i$。

该方法的代价如下。对于每个失效顶点 f，我们需要与其在 F-森林中的子节点数量相当的时间来创建集合 $\{c_1, \ldots, c_k\}$。然后，我们创建该集合的d 个副本，并对每个副本进行排序。这需要 $O(d \cdot k \log k)$ 时间，其中 k是 f在 F-森林中的子节点数量。由于每个失效顶点都要执行此操作，因此总成本为 $O(d^2 \log d)$。现在，对于每个失效顶点 f，以及每个作为 f祖先的内部连通分量 C'，我们需要 $O(\log n)$ 时间来找到线段 S，如上所述。因此，总成本还会增加 $O(d^2 \log n)$。现在，对于这个 f和 C'，以及每个 $i \in \{1, \ldots, d\}$，我们都必须像上面那样遍历列表 $\mathcal{C}_i$（同时执行常数时间的操作）。由于 $\mathcal{C}_i$ 的大小等于 f 在 F-森林中的子节点数量，因此总成本为 $O(d^3)$。

根据上述分析，我们可知，Line 8 的总成本为 $O(d^3 + d^2 \log n)$。[7] 因此，接下来只需对 Line 19 中执行二维范围查询的次数给出上界。为此，我们引入以下记号。设 f 为一个失效顶点，并令 $\mathcal{C}$ 表示其祖先中的内部连通分量的集合。于是，对于每个 $C' \in \mathcal{C}$，以及每个 $i \in \{1, \ldots, d\}$，我们令 $\mathcal{S}_i(C')$ 表示 $L_i(f)$ 的极大线段集合，这些线段

[7] 在字长为 $O(\log n)$ 的 RAM 机器上，我们可以使用 van Emde Boas 树，将上述二分搜索作为前驱/后继查询来执行，从而将“$\log n$”因子降低为“$\log \log n$”。

由以下对象的悬挂子树的根组成 f 其低点$_i$ 位于 C'中。因此可以看出，在处理 f 期间（即第 **for** 行的外层循环1中），第19 行中的二维范围查询数量恰好为 $\sum_{C'\in\mathcal{C}}\sum_{C}\sum_{i\in\{1,\dots,d\}}|\mathcal{S}_i(C')|$ $(*)$，其中第二个求和是对内部连通分量 C 进行的，这些分量是 f 的祖先且是 C'的后代。

现在固定一个 $i\in\{1,\dots,d\}$。对于每个 $C'\in\mathcal{C}$，令 $S_i(C')$ 表示 $L_i(f)$ 中由 f 的子节点构成、且其低$_i$ 点位于 C' 的极大线段。注意，$\mathcal{S}_i(C')$ 中的每条线段都完全包含在 $S_i(C')$ 内。由于 $\mathcal{C}$ 中的内部连通分量两两不相交，因此 $\{S_i(C')\mid C'\in\mathcal{C}\}$ 中的线段也两两不相交，其总数因而至多为 $|\mathcal{C}|\le d$。由于失效顶点的数量为 d，f 中作为失效顶点祖先的子节点数量至多为 d。正是这些子节点的存在，可能迫使进一步划分 $\{S_i(C')\mid C'\in\mathcal{C}\}$ 中的线段，以得到 $\bigcup\{\mathcal{S}_i(C')\mid C'\in\mathcal{C}\}$。但每个这样的子节点都会将一条针对某个 $C'\in\mathcal{C}$ 的线段 $S_i(C')$ 最多分成两条子线段。（回顾上文关于第 8 行实现的分析。）因此，为了得到 $\bigcup\{\mathcal{S}_i(C')\mid C'\in\mathcal{C}\}$，$\{S_i(C')\mid C'\in\mathcal{C}\}$ 中的线段至多需要划分 d 次。于是，我们有 $\sum_{C'\in\mathcal{C}}|\mathcal{S}_i(C')|\le d+d=O(d)$。这意味着 $\sum_{i\in\{1,\dots,d\}}\sum_{C'\in\mathcal{C}}|\mathcal{S}_i(C')|=O(d^2)$，因此表达式 $(*)$ 可由 $O(d^3)$ 界定。由于这对每个失效顶点 f 都成立，因此第 19 行中二维范围查询的数量可由 $O(d^4)$ 界定。

□

3.5 回答查询

假设我们已经为 $\mathcal{R}$ 的内部连通分量构造了连通性图 $T\setminus F$，并计算出了其连通分量。因此，给定两个内部连通分量 C和 C'，我们可以在常数时间内确定 C和 C'是否在 $G\setminus F$中连通，只需检查 r_C 和 $r_{C'}$ 是否属于 $\mathcal{R}$ 的同一个连通分量（见引理 3.8）。

现在，令 x,y为 $V(G)\setminus F$中的两个顶点。为了确定 x,y是否在 $G\setminus F$中连通，我们尝试将 x,y 替换为 $T\setminus F$ 的内部连通分量的根，然后将查询归约到这些根。具体而言，如果 x （分别地，y）属于 $T\setminus F$的一个内部连通分量C，那么 x,y之间的连通性与 r_C,y（分别地，x,r_C）之间的连通性相同。否则，如果 x（分别地，y）属于 $T\setminus F$的一个悬挂子树 H，那么我们尝试寻找一个通过回边与 H 连通的内部连通分量。如果存在这样的内部连通分量 C，那么我们可以将 x（分别地，y）替换为 r_C。否则，x,y在 $G\setminus F$中连通，当且仅当它们属于 $T\setminus F$的同一个悬挂子树。该思想如算法 3 所示。

命题 3.11。给定两个顶点 x，y在 $V(G)\setminus F$中，算法 *3*可以正确判断 x，y是否在 $G\setminus F$中连通。算法 *3*的运行时间为 $O(d)$。

证明。为了证明正确性，我们只需处理 x 位于 $T\setminus F$ 的悬挂子树 H 中的情况（第 6 行）。在这种情况下，我们只需检查 $G\setminus F$ 中是否存在一条边，将 H 与 $T\setminus F$ 的另一个连通分量连接起来。因此，根据引理 3.4，我们需要检查 H 是否通过一条回边与 $T\setminus F$ 的某个内部连通分量连通。如果存在这样的内部连通分量，令 C 为其中最低的一个。由于 $p(r_H)$ 是一个失效顶点，且 H 的祖先中失效顶点的数量不超过 d，因此至少存在一个 $i\in\{1,\dots,d\}$，使得 low $_i(r_H)$ 位于 C 中。于是，在 $G\setminus F$ 中对 x、y 的连通性查询，等价于对 low $_i(r_H)$ y 的查询。由于 low $_i(r_H)$ 属于内部连通分量 C，算法最终会终止，并给出正确结果。

否则，如果不存在通过回边与 H 相连的内部连通分量，那么 H 就是 $G\setminus F$的一个连通分量，因此 y 在 $G\setminus F$中与 x 相连，当且仅当 y 也位于 H内。现在，如果**针对**的循环（第 8 行）已搜索完毕，并且$low_d(r_H)$ 不存在，或者 $low_d(r_H)$ 是一个失效顶点，那么我们可以确定，不存在将 H 与图 $G\setminus F$的其余部分相连的回边。

算法 3： query(x, y)

如果 x位于一个内部连通分量中，且 y 位于一个内部连通分量中 C' **则**
    if r_C 在 $\mathcal{R}$ 中与 $r_{C'}$ 连通**则返回 true**
    返回 false
结束
// at least one of x, y lies in a hanging subtree
若 x位于悬挂子树 H **中，则**
    // check whether H is connected with an internal component through a back-edge
    对于 $i \in \{1, \ldots, d\}$，**执行**
        若$low_i(r_H) \neq \perp$ 且 $low_i(r_H) \notin F$**则**
            返回 query($low_i(r_H)$ y)
        end
    end
    // there is no internal component that is connected with H in $G \setminus F$
    若 y 位于 H，**则返回 true**
    返回 false
结束
返回 query(y, x)

现在我们将证明 $O(d)$ 的时间界。首先，给定一个顶点 $x \notin F$，我们可以通过找到最近的、作为 x 祖先的失效顶点 f，来确定包含 x 的 $T \setminus F$ 连通分量。通过找到作为 x 祖先的最大失效顶点，可以在 $O(d)$ 时间内完成此操作。如果不存在这样的顶点，那么 x 属于以 r 为根的内部连通分量。否则，包含 x 的 $T \setminus F$ 连通分量 C 的根，是 f 中作为 x 祖先的子节点。通过进行层级祖先查询，找到深度等于 $\mathsf{depth}(f) + 1$ 的 x 祖先，即可在常数时间内确定此子节点。现在，给定 $T \setminus F$ 的一个连通分量 C 的根 r_C，我们可以在 $O(d)$ 时间内，通过检查是否存在 r_C 的后代失效顶点，来判断 C 是内部连通分量还是 $T \setminus F$ 的悬挂子树。最后，如果我们有按递增顺序排列的失效顶点列表，就可以在 $O(d)$ 时间内执行第 8 行中的搜索。（在更新阶段，我们可以额外花费 $O(d \log d)$ 来完成排序。）然后，我们可以轻松地在 $O(d)$ 时间内检查是否存在一个 $i \in \{1, \ldots, d\}$，使得 $\mathsf{low}_i(r_C) \in F$，因为列表 $\mathsf{low}_1(r_C), \ldots, \mathsf{low}_d(r_C)$ 也是按递增顺序排列的。 □

致谢。 我谨向我的导师 Loukas Georgiadis 致谢，感谢他对本稿提出的宝贵意见。

参考文献

[1] Michael A.Bender 和 Martin Farach-Colton。《层级祖先问题的简化》。理论计算机科学，321(1):5–12，2004 年。 doi:10.1016/j.tcs.2003.05.002。

[2] Timothy M. Chan、Kasper Green Larsen 和 Mihai P˘atra¸scu。重新探讨随机存取机器上的正交范围查询。载于第*27*届*ACM*计算几何研讨会论文集，第1–10页，2011年。
doi:10.1145/1998196.1998198。

[3] Mark de Berg、Otfried Cheong、Marc J. van Kreveld 和 Mark H. Overmars。《计算几何：算法与应用》第3版。Springer，2008年。网址： https://www.worldcat.org/oclc/227584184。[4] Reinhard Diestel。《图论》第4版，数学研究生教材第173卷。Springer，2012年。[5] Ran Duan 和 Seth Pettie。针对顶点失效图的连通性预言机。《SIAM计算杂志》，49(6):1363–1396，2020年。 doi:10.1137/17M1146610。[6] Yaowei Long 和 Thatchaphol Saranurak。近乎最优的确定性顶点失效连通性预言机。载于第63届IEEE计算机科学基础年会（FOCS），第1002–1010页，2022年。 doi:10.1109/FOCS54457.2022.00098。[7] Hiroshi Nagamochi 和 Toshihide Ibaraki。在线性时间内寻找k连通图的稀疏k连通生成子图的算法。《Algorithmica》，7(5&6):583–596，1992年。doi:10.1007/BF01758778。[8] Michal Pilipczuk、Nicole Schirrmacher、Sebastian Siebertz、Szymon Torunczyk 和 Alexandre Vigny。顶点失效情况下带连通性的第一阶逻辑算法与数据结构。载于第49届国际自动机、语言与程序设计研讨会（ICALP），LIPIcs第229卷，第102:1–102:18页，2022年。 doi:10.4230/LIPIcs.ICALP.2022.102。[9] Robert Endre Tarjan。深度优先搜索与线性图算法。《SIAM计算杂志》，1(2):146–160，1972年。 doi:10.1137/0201010。