软件 CIDR 路由查找算法学习报告

——以 Luleå 算法为例

王逸松　于纪平　谭闻德

{wys19, yjp19, twd19}@mails.tsinghua.edu.cn

2019 年 10 月

目录

引言

Luleå 算法[1]像一件精美的艺术品，二十年来一直在软件 CIDR 路由查找算法中大放光彩。然而我们认为，对于 Luleå 算法的学习，不能只是“知其然”，更要“知其所以然”。就像我们在阅读数学定理的证明（尤其是利用到数学归纳法的证明）时，不能只是弄明白证明的过程、验证证明的正确性就学习完毕了，更重要的是理解它背后的思维方式。或者通俗些讲，要弄明白艺术品般精美的原理背后的草稿纸上写着什么，才算是理解了 Luleå 算法。

因此，本文将以另一种途径研究 Luleå 算法的“来历”，试图探究作者的思维方式，得到了一种类似 Luleå 的路由查找算法。最后，本文将详细介绍 Luleå 算法，以及比较我们得出的算法与 Luleå 算法的异同。

重新思考软件 CIDR 路由查找算法

本文讨论的范围是软件 IPv4 路由查找算法。虽然所有的实验都在 x86 架构上进行，但是我们并没有将范围限定在 x86 架构之内。一般而言，具有主存—缓存（片上或片外）层次结构的单核心处理器均适用于本文的讨论。

本文首先考虑两种路由查找算法，基于范围的路由查找算法，以及基于 Trie 树的路由查找算法。在进一步讨论之前，先给出 CIDR 路由查找算法的具体需求，以及评价算法好坏的方法。

算法需求与评价

对于软件 CIDR 路由查找算法，我们可以将其功能抽象为以下三种操作：

- 初始化：输入完整的路由表（每一项路由条目包括前缀、前缀长度以及下一跳信息），使软件进行预处理操作。
- 查找：输入目的地址，输出在路由表中查询匹配的最长前缀对应的下一跳信息，或者输出不存在对应路由表项的提示。
- 更新：对路由表增加、删除或修改某一项。

对于一种正确实现了以上功能的算法，我们可以采用以下不同的标准进行评价：

- 空间开销：算法使用的内存空间总量。
- 时间开销：算法初始化的时间、每次查找的时间、每次更新的时间。
- 各种其他方面，包括编程复杂度、理解难度等。

基于范围的路由查找算法

朴素的范围路由查找算法

在仅有软件 CIDR 路由查找算法的限制下，如果我们只需要考虑查找的时间效率，那么一个可行的思路是：由于 IPv4 地址仅有$2^{32} \approx 4.3 \times 10^9$种，我们可以直接存下所有可能的地址对应的下一跳信息，即可在每次请求到来时直接查表得到相应信息。

- 空间开销：为了存下整张表，共2^{32}个表项，假设每个表项 4 字节，共需2^{34}字节，即 16 GiB。
- 时间开销：初始化需要的时间较长，更新可能难以高效实现，但每次查找很快，仅需 1 次内存访问。

这个算法虽然看似具有极快的查找效率，但是有着巨大的空间开销，很难运用到实际当中。并且，这一较大的空间开销同时对于时间效率有负面影响。即使内存能够容纳 16 GiB 的表，这一算法也很难利用层次存储的性质，即难以利用缓存进行加速，因为目前的缓存无法完全容纳 16 GiB 的内容。注意到主存的访问延迟可能比缓存要高一个甚至几个数量级，所以即使是稳定的 1 次主存访问，也不见得是一个较优的方案。

二分的空间优化范围路由查找算法

上一节所描述的算法的一大问题是空间开销过大。我们可以通过如下方式减少空间使用量：考虑到大多数情况下，相邻两个 IP 地址在表中对应的下一跳应当很可能是相同的；换言之，很可能有连续一大段 IP 地址的空间对应着一个相同的下一跳信息。于是，我们完全可以抛弃2^{32}个项目的大表，转而记录形如“从某个地址 L 到另一个地址 R 对应的下一跳是某某 x”的信息。

所有这些信息所对应的区间$[L, R]$应当是两两不相交的。假如我们把不存在合法路由信息的地址也统一归到某个下一跳中，并且将所有这些信息按照 L 排序之后，对于相邻的两条信息，前一条信息的 R+1 必然等于后一条信息的 L。这意味着，我们并没有必要存储每一条信息的 R，仅需要从它的下一条信息的 L 推算而来即可。

处理完所有这些信息之后，对于每次查找请求，设该请求查询的地址为 x，则我们需要在这些信息中找到 $L \leq x$ 且 L 最大的一条信息，则这条信息的下一跳值就是我们所求的答案。而找到 $L \leq x$ 且 L 最大的一条信息这一步可以使用二分查找的方法解决。

- 空间开销：这与我们求出的信息的总数量有关，而这个总数量可能是我们难以准确计算出来的。不过，我们可以得出它的一个上界。假设路由表条目数为 n，则上述区间的数量不会超过 2n+1 个[i]，于是我们需要存储的信息总条数不会超过 2n+1。对于8×10^5个条目[ii]

[i] 每个路由条目对应的区间包含有两个端点，于是 n 个路由条目会产生 2n 个端点。在一个线段内部插入 2n 个点可以将此线段分割成至多 2n+1 个区间。

[ii] 这是 2019 年 9 月 IPv4 路由表条目的大致数量。

的路由表，需要存储的信息的数量不会超过1.6×10^6条。假设每条信息需要 8 字节进行存储，总存储量下降到了约 12.7 MiB。

- 时间开销：初始化需要的时间较长，更新可能难以高效实现；每次查找需要进行二分查找，即 O(log n)次内存访问。

位图优化的范围路由查找算法

上一节所描述的算法当中，一大问题是时间开销过大，复杂度从 O(1)变为了 O(log n)。是否有办法在时间复杂度仍为 O(1)的前提下压缩路由表占用的空间呢？在我们之前的分析当中，一个很好的性质是：相邻地址对应的下一跳很可能是相同的，不同的段数最多有 2n+1 段。

设在朴素算法当中，我们存储的完整预处理表为 nexthop[]，则我们开设一个辅助表 A[]：如果 addr 为零或者 nexthop[addr]与 nexthop[addr - 1]不同，则 A[addr]为 1，否则为 0。注意到辅助表 A[]的每一项仅可能为 0 或 1，我们可以用位图存储表 A[]。每个字节的宽度为 8 位，于是我们仅需$2^{32}/8$个字节，即 512 MiB 的空间即可存储 A[]。

辅助表 A 的意义在于：对于地址 addr，A[0] + A[1] + … + A[addr]的结果就是 addr 处于 nexthop[]表的第几段。现在，我们仅需存储每一段对应的路由信息 nexthop’[]，那么 nexthop’[A[0] + A[1] + … + A[addr]]就是该地址对应的路由条目。

已知 addr，现在考虑 A[0] + A[1] + … + A[addr]的快速算法。一个朴素的想法就是求 A[]的前缀和 S[]，其中 S[k] = A[0] + A[1] + … + A[k]。然而，如果我们存下所有 addr 对应的 S[]值，就失去了优化空间的意义。我们这里转而存储 T[] = S[0]、S[w]、S[2w]、S[3w]……的值，即每隔 w 个值存储一个前缀和，其中 w 是机器的字长。这样，每当我们想求解 S[]的某一位 S[x]时，设 x=aw+b（其中 a 和 b 是非负整数且 b<w），我们可以直接查找 T[a] = S[aw]，然后遍历 A[]的 b 个元素 A[aw + 0]、A[aw + 1]、…、A[aw + b - 1]并将它们相加即可得到 S[x]的值。

由于 b<w，遍历连续的 b 个元素并进行求和的操作可以快速进行。一般来说，这个问题可以在移位之后转化为 popcount 问题。popcount 问题是指统计一个整数对应的二进制表示中 1 的个数，这个问题已有成熟的软件方法（查表法、分段查表法）和硬件方法（例如 x86 架构在 SSE4.2 后支持 popcnt 指令，Arm 架构在 Neon 后支持 vcnt 指令，System/Z 架构支持 popcnt 指令）。

综上所述，我们可以通过一次访存与一次 popcount 指令得到任意一个 S[]的值，再通过该值在 nexthop’[]表中查找对应的下一跳地址。

- 空间开销：预处理表 A[]所用的空间为 512 MiB；注意到表 T[]中每一项需要占用 4 字节，表 T[]所用的空间为 4*4/w GiB，表 nexthop’[]所用的空间约 6.4 MiB（参考上一节描述的算法，这一空间为上一节所用的表的空间的一半）。对于 w 至少为 32 的情况下，表 S[]所用的空间不超过 512 MiB，总空间开销至多约 1 GiB。
- 时间开销：初始化需要的时间较长，更新可能难以高效实现；每次查找需要 3 次内存访问，包括一次表 T[]的访问、一次表 A[]的访问、一次表 nexthop’[]的访问。然而，对于表 T[]与表 A[]的访问具有局部性，一般情况下可以合并为一次访问；对表 nexthop’[]的访问范围相

对较小，在现代处理器中可以利用缓存的快速访问性质。虽然是 3 次内存访问，但是其中两次可以合并，另一次为缓存的访问，所以总的访存延迟约为 1 次内存访问时间加上 1 次缓存访问时间。

基于 Trie 树的路由查找算法

利用 Trie 树存储路由条目以及进行路由查找是另一种直观的想法。Trie 树是用来存储一个字符串集合的数据结构。如果我们将 IPv4 地址视为长度为 32 的 01 串，则可以使用 Trie 来存储所有的路由表，并在其上面进行最长前缀匹配。

- 空间开销：假设路由表项的条目数为 n，Trie 树中的节点数上界为 33n，这也是该算法的空间复杂度。对于 $n=8\times10^5$ 这个上界约为 2.6×10^7。然而，考虑到不超过 23 层的节点至多有 $2^{23}\approx8\times10^6$ 个，超过 23 层的节点至多有 $10n=8\times10^6$ 个，这个上界可以减少为 1.6×10^7（实际上，考虑到路由表的多数表项的前缀长度都比 32 小，这个上界仍然很宽松）。设每个节点存储左右孩子指针各 4 字节与其他信息 4 字节共 12 字节，则总空间约为 192 MiB。
- 时间开销：从查询算法可以看出，该算法在最坏情况下需要访问 32 次内存。不过 Trie 树中离根较近的节点更容易被访问，所以 Trie 树本身的结构可以充分利用层次存储器的优势。实际的 32 次内存访问会分布在各个层次中，一定程度上减轻内存的压力，但内存的访问次数仍然不小。

基于 Trie 树的位图优化的路由查找算法

基于上面的讨论，位图优化的范围路由查找算法空间需求大，但仅需一次内存访问即可查询到路由信息；而基于 Trie 树的路由查找算法空间开销较小，但需要若干次内存访问才可以得到答案。现在，考虑将上述两种算法结合，尝试结合两种算法的优点。

注意到我们可以对 Trie 树的前几层进行“压扁”操作（或称对某一层进行“切割”操作），把它们变为一层，对于该层即可使用类似位图优化的方法来查找下一跳信息或者下一级子树根节点的指针。这样，我们相对于 Trie 树增加了空间使用，但是减少了内存访问次数。为了达到最优解，需要确定的参数是，压扁多少层。

根据目前 IPv4 路由表的统计数据，前缀长度为 24 的路由表项约占一半，因此有必要在第 24 层进行一次切割操作。如果在第 24 层的后几层进行切割，前缀长度为 24 的路由表项会展开为更低层的若干表项，浪费存储空间；如果在第 24 层的前几层进行切割，为了查找前缀长度为 24 的路由表项包含的地址，需要多若干次内存访问，增加了路由查找时间。此外，特别地，若在 16 层或之前进行切割，表 T[]中每一项的大小可以减小为 2 字节。例如，若压扁前 16 层，表 A[]所用的空间为 8 KiB；注意到表 T[]中每一项需要占用 2 字节，表 T[]所用的空间为 $64\times2/w$ KiB，对于 $w = 32$，总空间约为 8 KiB + 4 KiB = 12 KiB，下一级子树树根的指针数组的大小则与路由表实际情况有关。

几次切割操作结合起来成为一个“划分”。不同的划分方式可以造就不同的算法，本文将这类算法的划分方式记为“$x_1+x_2+\ldots+x_k$ 型”，表示该算法共将 Trie 树切割为 k 大层，第一大层消耗 x_1 位 IP 地址，第二大层消耗接下来 x_2 位 IP 地址，最后一大层消耗剩下 x_k 位 IP 地址。例如，上节基于 Trie 树的路由查找算法为（32 个 1）“1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 型”，位图优化的范围路由查找算法为“32 型”，本文将详细介绍的 Luleå 算法属于“16+8+8 型”。

Luleå 路由查找算法

正如上节最后所述，Luleå 算法共分为三层，第一层为 16 bit，第二层与第三层各 8 bit。

在第一层中，位图中每 16 bit 对应一个 code word，其中包含：

- 一个 10 bit 的索引，表示是哪一种 16 bit；
- 一个 6 bit 数，表示该 16 bit 之前的 bits 中出现 1 的次数。

下面通过说明这 16 bit 的有效取值只有 678 种，来说明 code word 的索引仅需 10 bit。

若记长度为 2^n 的 code word 的合法种类数为 a_n，那么 a_n 满足下式：

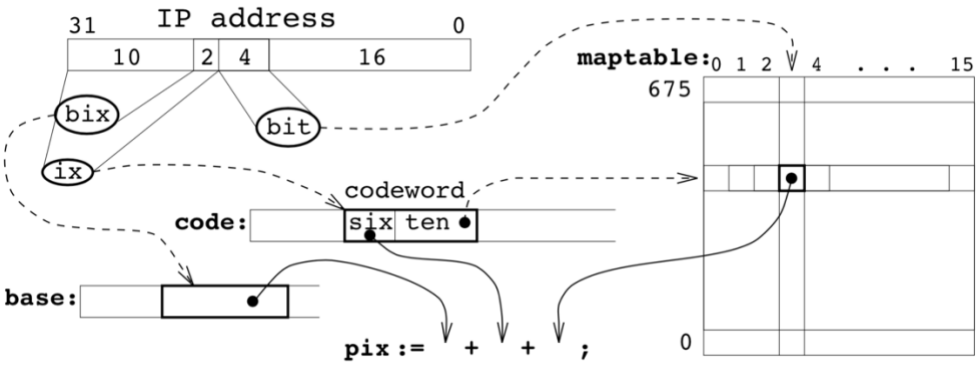
$a_0 = 1, a_n = 1 + a_{n-1}^2 \ (n>0)$

$a_0 = 1$ 时显然成立；而对于 n>0，长度为 2^n 的 code word 要么是首位为 1 其余位为 0，这部分方案数为 1，要么可以分为两个长度为 2^{n-1} 的独立的 code word，根据乘法原理，这部分方案数为 a_{n-1}^2，再根据加法原理可得上面 a_n 的表达式。

利用这 10bit 的索引就可以在一个叫做 map table 的表中查到该索引对应的 16bit 中某一位对应的前缀和信息。

由于前述 code word 的 6 bit 数的表示范围仅为 0~63，因此 4 个 code word 之后，该字段便有可能溢出。Luleå 算法的做法是给每 64 bit 存储一个 base index，记录该 64 bit 组之前的 bits 出现的 1 的次数。一个 base index 为 16bit，范围 $0\sim2^{16}-1$，足够记录当前 64 bit 组中该 16 bit 之前的 bits 中出现 1 的次数。

结合上面的介绍，Luleå 算法第一层的工作方式如下所示：



首先查询 base index，然后查询 code word 中的 6bit 数，最后查询 map table 对应项目，它们相加得到的和就是该地址对应路由条目的索引号。

事实上，经过“基于 Trie 树的位图优化的路由查找算法”一节的讨论，我们可以发现 Luleå 算法中 code word 以及 base index 的设计就是为了实现快速计算“A[]的前缀和 S[]”的目的。具体而言，其中 code word 以及 map table 的效果就是为了快速解决“位图优化的范围路由查找算法”一节介绍的 popcount 问题，base index 就对应 T[]。然而，值得注意的是，在 Luleå 算法设计时还没有 64 位计算机，当时的计算机也还没有 popcnt 等指令。作者在当时如此巧妙的设计，其巧妙利用了 Trie 树性质将 16bit 的位图编码为 10bit 的方法仍然令现在的读者叹为观止。

Luleå 算法对于第二层以及第三层的处理方式相同：

- 对于 1~8 个子树（或下一跳信息）的稀疏情况：每个子树直接存储 8bit 索引、16bit 子树指针；
- 对于 9~64 个子树的稠密情况：code word 中的 6bit 可以存下 1 的个数，因此只需一个 base index 用于存放该子树的子树指针相对于所有子树指针的起始索引；
- 对于 65~256 个子树的非常稠密的情况：类似第一层。

基于评测鸭的路由查找算法测试平台

评测鸭简介

在程序设计竞赛和训练中，评测是一个重要的环节。该环节主要测试选手程序的运行时间、内存使用和正确性。目前，评测由评测系统自动化地完成。而现有评测系统面临的一个重大挑战是，多次测试同一程序时，时间波动较大。

评测鸭[2]是一个为程序设计竞赛和训练打造的稳定精确评测系统。通过重新编写操作系统，消除了大部分不稳定因素，实现了稳定的评测。其特点主要有：

- 精确：计时精度达 1 纳秒[iii]，精确把握程序运行时间。
- 稳定：多次测试同一程序，时间误差不超过 1%+1 微秒[iv]，领先业界主流评测系统[v]一个数量级。
- 兼容：兼容 C、C++ 等主流程序设计语言[vi]，实现从其他评测系统的极速迁移。
- 自研：自主研发评测专用操作系统，掌握关键技术。
- 易用：简单易用的云评测服务，一分钟快速上手。

路由查找算法的抽象与在评测鸭上的移植

我们将路由查找算法抽象为一道程序设计竞赛风格的交互式题目，并发布在了评测鸭上，名为“测测你的路由器”[vii]。本题目具有以下特点：

- 由于在评测鸭平台上进行测试，我们会受到评测鸭的优点与局限性的影响：
  - 在 95%置信度范围内，不超过 1%+1 微秒的误差（在本题当中，由于实际的运行时间一般远大于 100 微秒，所以可以认为误差在 1%以内）。
  - 评测鸭是 x86 平台（32 位），我们此次作业仅能在 x86 平台（32 位）上测试软件路由查找算法，而无法测试在 64 位机器上运行的效率。事实上，我们也在常用的 x86-64 平台下使用 Linux 操作系统进行了测试，但是由于误差过大（在几十毫秒左右的运行时间下，相对误差可能高达百分之十几到几十），导致难以进行准确的分析。
  - 受到评测鸭对于语言支持的限制，测试者仅能使用汇编、C 或 C++语言编写代码。但是一般来讲，这些语言对于实现 x86 平台下的软件路由查找算法已经足够了。
- 时间限制为 30 秒，空间限制为 2048 MB。这是一个非常宽松的限制，因为典型的比较优的算法都会在 1 秒、128 MB 内完成执行；设置如此宽松的限制是为了方便调试与比较其他各种路由查找算法。
- 主要目的是测试静态路由表的查询开销，而不测试路由表的更新。
  - 测试者仅需要实现 `init`（初始化）与 `query`（查询）两个函数。不需要处理需要更新的情况。
  - `init` 的函数原型如下所示：

```
typedef struct {
    unsigned addr;
    unsigned char len;
```

[iii] 1 纳秒=10^{-9} 秒。
[iv] 使用评测鸭对多种排序算法程序测得，95%置信度的时间测量误差不超过 1%+1 微秒。
[v] 业界主流评测系统，是指洛谷、UOJ、大视野在线测评、Vijos 等知名度较高的评测系统。
[vi] 兼容语言和标准库的主要特性，暂不支持第三方库（如 Qt）。
[vii] https://duck.ac/problem/router32。

```
    char pad[3];  // Padding for memory alignment
    unsigned nexthop;
} __attribute__((packed)) RoutingTableEntry;

void init(int n, int q, const RoutingTableEntry *a);
```

- ◆ 在程序启动时 `init` 会被调用一次，接下来 `query` 会被调用 `q` 次，随后程序将结束。
- ◆ `n` 表示路由表项的数量，`q` 表示询问的数量，`a` 为一个长度为 `n` 的数组，表示各路由表项。
- ◆ 输入数据保证，所有路由表项已按照 `addr` 为第一关键字，`len` 为第二关键字升序排序。保证路由表中不存在两个 `addr` 和 `len` 分别完全相同的表项。

- ■ `query` 的函数原型如下所示：

```
unsigned query(unsigned addr);
```

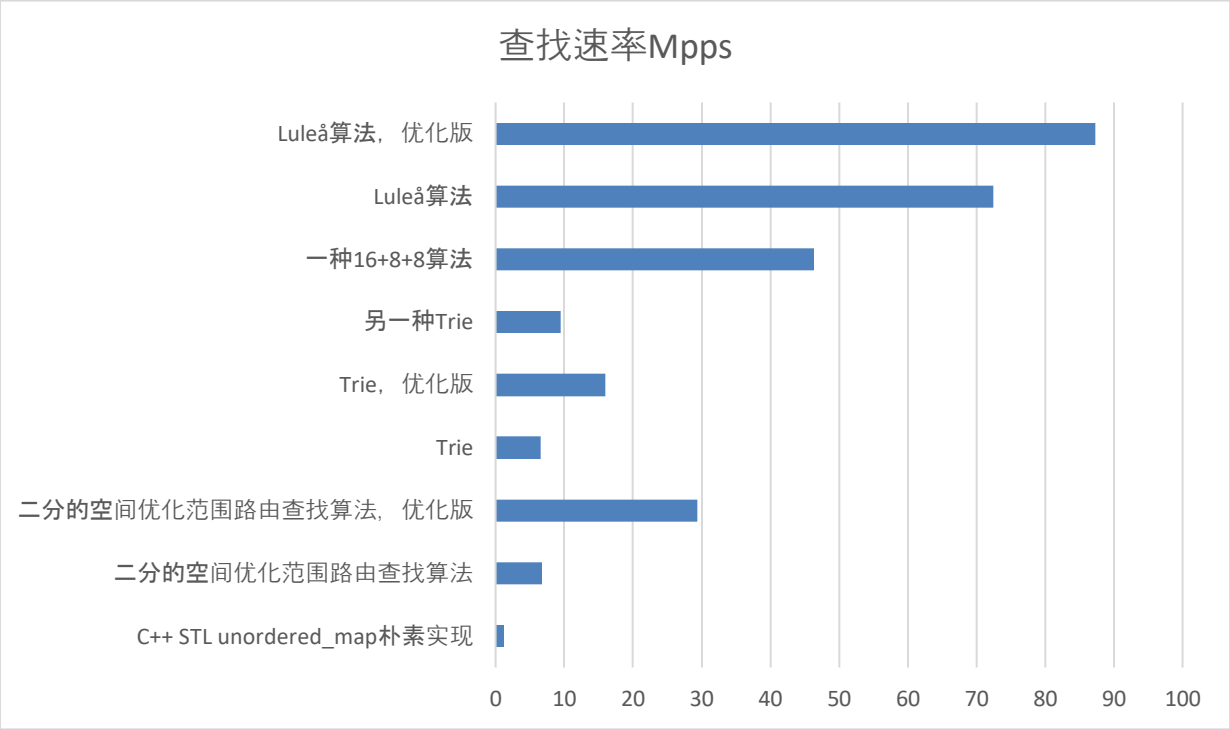
- ◆ 查询 `addr` 地址。若在路由表中能找到满足条件的表项，则返回其 `nexthop` 值，否则返回 0。

- 设有 4 个测试点，以便分别测试路由查找算法的建立与查询的开销。详细信息如下所示：
  - ■ 第 1 个测试点中$n = 1, q = 1$，用来测试程序初始化的时间。
  - ■ 第 2 个测试点中$n = 827088, q = 1$，用来测试路由表的建立时间。即，对于同一份程序，它在第 2 个测试点中的用时与在第 1 个测试点中的用时之差即为路由表的建立时间。
  - ■ 第 3 个测试点中$n = 827088, q = 1000000$，而第 4 个测试点中$n = 827088, q = 2000000$，用来测试路由表的查询时间。即，对于同一份程序，它在第 4 个测试点中的用时与在第 2 个测试点中的用时之差即为在路由表上进行 2000000 次查询所用的时间。
- 我们选取了真实的路由表进行本题的测试。除第 1 个测试点以外，其余测试点的路由表数据来自 University of Oregon Route Views Archive Project[3]。在本题当中，我们使用的是 2019 年 9 月 17 日的快照版本[viii]。
- 每次查询是独立、均匀随机生成的。这可能并不是最接近实际情况的生成方式，但是我们在完成此次作业的过程中没有找到更“合理”的生成方式，也没有找到直接可用的访问记录日志等资料，所以暂时采取了这种最简单的方式来生成。

[viii] http://archive.routeviews.org/oix-route-views/2019.09/oix-full-snapshot-2019-09-17-0600.bz2。

测试结果

在此报告完成时，我们收到了近 500 次提交（多数由本文作者完成），其中 153 次为有效提交，即正确完成了路由查找的功能。我们统计了各种算法的发送速率如下：



算法	查找速率 Mpps	对应 Gbps
C++ STL unordered_map 朴素实现[ix]	1.2247	0.8231
二分的空间优化范围路由查找算法[x]	6.7431	4.5317
二分的空间优化范围路由查找算法，优化版[xi]	29.3668	19.7358

[ix] https://duck.ac/submission/10256
[x] https://duck.ac/submission/10350
[xi] https://duck.ac/submission/10364

Trie[xii]	6.5345	4.3915
Trie，优化版[xiii]	16.0171	10.7642
另一种 Trie[xiv]	9.4984	6.3833
一种 16+8+8 算法[xv]	46.3598	31.1558
Luleå 算法[xvi]	72.4264	48.6737
Luleå 算法，优化版[xvii]	87.3086	58.6751

[xii] https://duck.ac/submission/10384
[xiii] https://duck.ac/submission/10450
[xiv] https://duck.ac/submission/10549
[xv] https://duck.ac/submission/10567
[xvi] https://duck.ac/submission/10602
[xvii] https://duck.ac/submission/10657

参考资料

[1] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. 1997. Small forwarding tables for fast routing lookups. SIGCOMM Comput. Commun. Rev. 27, 4 (Oct. 1997), 3–14. https://doi.org/10.1145/263109.263133

[2] 评测鸭，JudgeDuck，https://duck.ac/。

[3] University of Oregon Route Views Archive Project，http://archive.routeviews.org/。