高效 IP 路由查找实验

范子墨
2021K8009929006

一、 实验内容

a) 实现最基本的前缀树查找
b) 调研并实现某种 IP 前缀查找方案
   i. 参考文献：Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup, ACM SIGCOMM, 2015
   ii. 不限于上述文献，甚至可以不是前缀树结构
c) 基于 forwarding-table.txt 数据集(Network, Prefix Length, Port)
   i. 本实验只考虑静态数据集，不考虑表的添加或更新
   ii. 以前缀树查找结果为基准，检查所实现的 IP 前缀查找是否正确
      1. 可以将 forwarding-table.txt 中的 IP 地址作为查找的输入
   iii. 对比基本前缀树和所实现 IP 前缀查找的性能
      1. 内存开销、平均单次查找时间

二、 文献阅读

对于 PPT 中提到的参考文献，文献中提到了一个 Poptrie 方法，Poptrie 方法在树的构建、查找算法、树的更新上都进行了改进，这其中也包含了在 PPT 中提到的方法。

在 Poptrie 方法中，首先是查找算法，与二叉树类似，关键之一是利用 CPU 指令来计算位串中 1 和 0 的个数，即进行"population count"。这些计数分别用作后代节点和叶子节点的间接索引。

而后在构建好树后，可以用叶子位向量进行压缩。前面小节中描述的前缀扩展产生了许多重复和冗余的叶子。在普通的 2^k-ary 多路 trie 中，一个相同的 FIBentry -try 对应于一个较短的前缀，可以在一个内部节点内冗余地跨到多个叶子，最多可达 2k - 1 个叶子。比如说，当 k = 6 时，一个内部节点可以有一个值为 a 的下一跳和 63 个值为 B 的下一跳，在它的 64 长度的叶子数组中。这样，冗余的叶子可能消耗大量的内存。因此 Poptrie 引入位向量 leafvec，可以将上述的冗余叶子压缩为只有一个叶槽，压缩存储空间。

而后是直接指向，真实数据大多数前缀都分布在前缀长度从/11 到/24 的范围内。这意味着，对于大多数 IP 地址，任何树结构的查找算法总是需要遍历至少部分内部节点才能到达叶子节点，从而消耗一些存储空间。因此可以设置一定的最高有效位作为数组的索引，即“直接索引”，用键地址 n 的最高有效 s 位的值作为直接索引。通过访问顶级数组中的第 n 个元素，它能够直接跳转到相应的 FIB 项或内部节点。

最后是增量更新，即让 IP 转发过程继续引用当前(即旧的)FIB，而更新的 FIB 的构建正在进行中。更新完成后，通过使用原子指令更改 FIB 的 指针或索引，将当前的 FIB 切换到新的 FIB。

三、 基本前缀树

1、 构建前缀树

对于构造前缀树，只需阅读条目然后将条目插入到已有的树上。由于树为二叉树，因此一个节点有两个子树，然后依次向下。

```
typedef struct TrieNode {
    struct TrieNode *children[2];
    uint32_t ip;
```

```
    uint32_t prefix;
    int has;
    int port;
}TrieNode;
```

在 insert_node 函数中进行节点插入，构造 cur_node 和 next 两个节点分别指向当前节点和下一个节点。使用 while 循环不断进行迭代，如果当前节点小于要插入节点的前缀长度，则判断下一个 bit 是 0 还是 1，以决定 next 节点，如果 next 节点是 null 则将 next 节点初始化。注意这里的有效位 has 不能设为 1，因为还没到达最长前缀匹配，当到达最长前缀匹配时跳出循环，而后将 cur_node 赋为 1。

```
while (cur_node && cur_node -> prefix < prefix)
```

2、 查找前缀树

思路与条目插入类似，对测试文件里所有的条目调用 find_ip 函数。如果当前节点 has 位有效且前缀相同，则匹配，而后向子树继续匹配。

```
while(cur) {
        if (cur -> has && cur -> ip == get_prefix(ip, cur -> prefix))
            match = cur;
        cur = cur -> children[get_bit(ip, 31 - cur -> prefix)];
    }
```

四、 优化前缀树

整体设计思路来源于 PPT 上的两种优化方法以及参考文献中的内容，首先构建一个多 bit 前缀树，每次匹配 2bit。同时考虑到参考文献中提到的直接指向，因此将首次匹配设置为 16bit。此外，在实现了上述前缀树后，想进一步加快速度，压缩节点，但是在运行过程中仍然存在问题，因此这里只做大致思路记录。

参考基础前缀树的数据结构，优化前缀树增加 is_odd 和 compress 变量，分别作为 2bit 前缀树匹配和压缩节点时的一个参考量。

1、 构建优化前缀树-多 bit

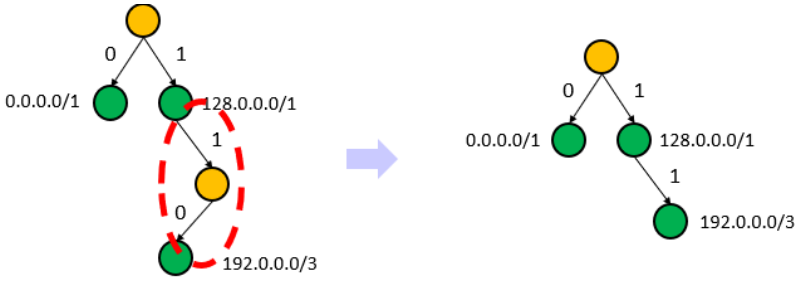
首先构建一个优化树，将根节点的子节点设为 65536 个，并将数据结构中各项都初始化，这里注意 has=0 以及 prefix=PREFIX_LEN。而后插入节点时，处理分为 prefix<16 和 prefix ≥16 处理。

当 prefix＜16 时，由于原 prefix 长度为 16，因此要将这 16 位拆分为两部分，一部分是插入条目的 ip，一部分是剩余不在 prefix 长度中的前缀。而当进行匹配时，以插入条目 ip 为开头的所有一层节点都要赋上条目的值，否则可能存在匹配却查找失败。因此设定初始值和结束值，将这其中的所有节点均进行赋值。同时，在有更长且小于 16 的条目时对原节点的值进行覆盖。

当 prefix≤16 时，首先进行 2bit 一组进行插入，如果 prefix 长度为奇数的话，同样插入，而且设置 is_odd 为 1。同时，如果 prefix 长度为偶数需要覆盖原来的一个 prefix 长度为基数的条目的话，只需将 ip 值和 is_odd 改变即可。

```
else if(next -> is_odd == 1 && next -> prefix == prefix){
            next -> ip = get_prefix(ip, cur_node -> prefix + 2);
            next -> is_odd = 0;
        }
```

2、 构建优化前缀树-压缩中间节点（未实现）



判断该节点可以删除在于其子节点有且只有一个节点不是 NULL，且该节点 has=0，因此只需使用函数不断迭代即可。

```
if(root == NULL){
        return;
    }
    for(int i=0;i<4;i++){
        if(root->children[i] == NULL){
            continue;
        }
        if (root->children[i]->compress == 3 && root->children[i]->has == 0) {
            // Combine children using bitwise OR
            TrieNodeadv *cur_node = root->children[i];
            root ->children[i]= root->children[i]->children[0] ?   root->children[i]->children[0] :
                                    (root->children[i]->children[1] ? root->children[i]->children[1] :
                                    (root->children[i]->children[2] ? root->children[i]->children[2] :
root->children[i]->children[3]));

            free(cur_node);
        }
        search(root->children[i]);
    }
```

虽然删除了一些节点，但是树的查找路径没有变，之前的最长前缀查找依旧可以使用，无需更改。

经过更改 main 函数，将所有输出结果错误的条目打印。

```
if(port != port_vec[i]){
        printf("%dport_correct:%d   false: %d\n",i,port,port_vec[i]);
       //fclose(fp);
        //return false;
}
```

发现仅有一条错误。

```
leona@leona-virtual-machine:~/CN/10-ip_lookup$ ./ip_trie_tree
Constructing the basic tree......
Reading data from basic lookup table......
Looking up the basic port......
Constructing the advanced tree......
Reading data from advanced lookup table......
Looking up the advanced port......
31293 port_correct:5  false: 3
Dumping result......
basic_pass-1
basic_lookup_time-146605us
advance_pass-1
advance_lookup_time-48917us
leona@leona-virtual-machine:~/CN/10-ip_lookup$
```

但目前还未找到是什么问题。

3、 查找前缀树

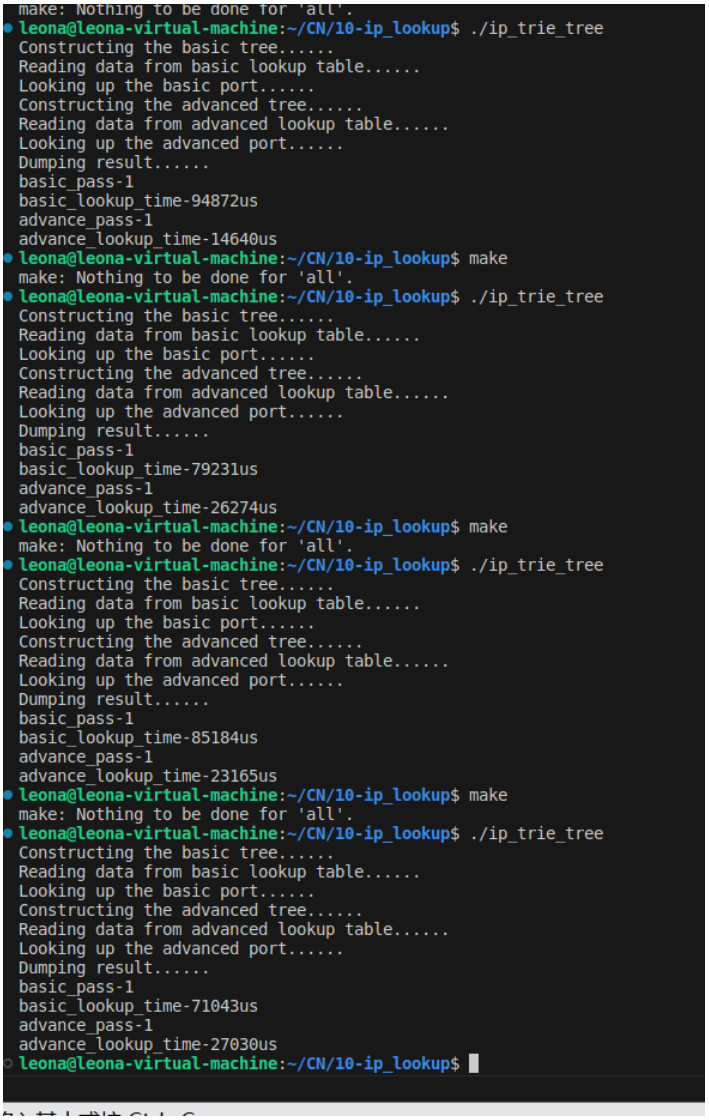
查找前缀树从一层节点开始寻找，而后两位两位进行匹配，与基本前缀树查找类似。

4、 可能优化方向

在我已有的代码上，除了实现以上说的压缩节点，根据参考文献所说，如果我将每次匹配 bit 数再次增大，可以添加 leafvec 进行压缩。

五、 优化结果

首先是第一次匹配长度到底多少的问题，我最后选择了 16bit，因为在更改第一次匹配长度时，发现 16 的时候结果比较好，因此设为 16bit。

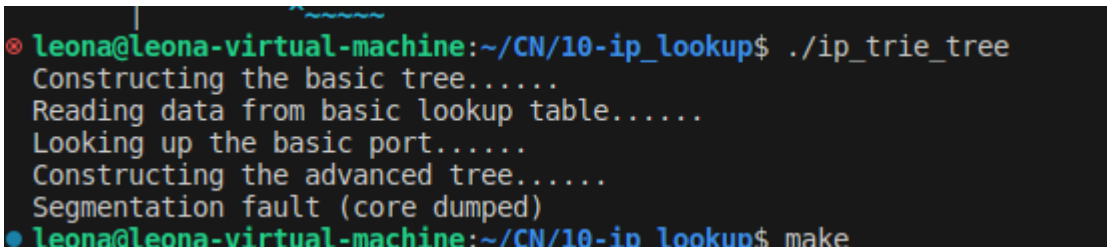


而在不加入压缩节点的情况下，速度已经提升一半。（不知道为什么，我的虚拟机运行速度很慢）

```
leona@leona-virtual-machine:~/CN/10-ip_lookup$ ./ip_trie_tree
Constructing the basic tree......
Reading data from basic lookup table......
Looking up the basic port......
Constructing the advanced tree......
Reading data from advanced lookup table......
Looking up the advanced port......
Dumping result......
basic_pass-1
basic_lookup_time-156473us
advance_pass-1
advance_lookup_time-53884us
```

六、 遇到的问题

1、数组越界



由于本实验中大量使用指针，因此经常出现 core dumped 的问题，因此对于使用结构体的指针，要首先检查是否是 NULL。