实验报告

高效 IP 路由查找实验

一、实验内容

了解 IP 路由查找效率的实际需求，学习经典的前缀树 IP 路由查找算法以及其优化方法，实现最基本的前缀树查找。调研并实现某种高级 IP 前缀查找方案。最后对两种方案的内存开销、平均查找时间进行对比分析。

二、实验流程

1. 实现最基本的前缀树 IP 路由查找。
2. 调研并实现某种高级 IP 前缀查找方案。
3. 对比分析两种方案的内存开销、平均查找时间。

三、实验结果及分析

（一）前缀树 IP 路由查找

1、关键数据结构

树节点数据结构如下，数据部分由网络号 net，掩码长度（前缀长度）prefix_len，端口号 port_id，以及标记节点类型的 type 组成，type 分为两种内部节点 INTERNAL 和匹配节点 MATCH。然后是用于构建树结构的指针，指向左右子节点和父节点，父节点指针用于回溯。

```
typedef struct node {
    int net;
    int prefix_len;
    int port_id;
    int type;
    struct node* left;
    struct node* right;
    struct node* parent;
} TreeNode;
```

路由表项结构如下，包括 ip 数值、掩码长度（前缀长度）prefix_len，端口号 port_id。

```
typedef struct {
    uint32 ipv4;
    int prefix_len;
    int port_id;
} RouteEntry;
```

2、关键函数

与前缀树相关的函数有如下 5 个。其中 Tree_Init 负责初始化根节点，Tree_Destroy 负责清空整棵树并释放内存，Tree_Add_Entry 负责向前缀树增加一个路由表项，Tree_Add_Node 负责在树内部增加一个节点，Tree_Lookup 负责根据 ip 查询对应端口。

```
void Tree_Init(TreeNode *root);
void Tree_Destroy(TreeNode *node_now);
void Tree_Add_Entry(RouteEntry *entry, TreeNode *root, long long *memory);
TreeNode *Tree_Add_Node(RouteEntry *entry, int prefix_len, int mask, TreeNode *parent);
int Tree_Lookup(int ipv4, TreeNode *root);
```

Tree_Add_Entry 根据所给路由表项的前缀长度，从根节点按位构建树，如果遇到想要访问的节点不存在，则建立该节点。最后对深度等于前缀长度处的节点标记为匹配节点 MATCH，路径中的节点标记为内部节点 INTERNAL。

```
int mask = 0x80000000;
uint32 prefix_bit = 0x80000000;

int add_num = 0;
for (int i = 1; i <= entry->prefix_len; i++) {
    if ((entry->ipv4 & prefix_bit) == 0) {
        if (node_now->left == NULL) {
            node_now->left = Tree_Add_Node(entry, i, mask, node_now);
            *memory += sizeof(TreeNode);
            add_num++;
        }
        node_now = node_now->left;
    }
    else {
        if (node_now->right == NULL) {
            node_now->right = Tree_Add_Node(entry, i, mask, node_now);
            *memory += sizeof(TreeNode);
            add_num++;
        }
        node_now = node_now->right;
    }
    mask = mask >> 1;
    prefix_bit = prefix_bit >> 1;
}
```

Tree_Lookup 根据给定 ip 进行查询，从树的根节点开始遍历，逐位匹配。为了获取最大长度的匹配，总是向下匹配到无相应子节点为止。当匹配到无子节点时，检查当前节点的类型，如果是匹配节点 MATCH，说明当前节点就是最长匹配，返回当前节点的 port_id。如果是内部节点 INTERNAL，则需要向后回溯，直到找到一个匹配节点为止，此时的匹配节点就是最长匹配，返回该节点的 port_id。

```
uint32 prefix_bit = 0x80000000;
int i, port;
for (i = 1; i <= 32; i++) {
    if ((ipv4 & prefix_bit) == 0) {
        if (node_now->left == NULL) {
            if (node_now->type == MATCH) {
                return node_now->port_id;
            }
            else {
                while (node_now->parent) {
                    if (node_now->parent->type == MATCH) {
                        return node_now->parent->port_id;
                    }
                    node_now = node_now->parent;
                }
                return -1;
            }
        }
        node_now = node_now->left;
    }
    else {
        if (node_now->right == NULL) {
            if (node_now->type == MATCH) {
                return node_now->port_id;
            }
            else {
                while (node_now->parent) {
                    if (node_now->parent->type == MATCH) {
                        return node_now->parent->port_id;
                    }
                    node_now = node_now->parent;
                }
                return -1;
            }
        }
        node_now = node_now->right;
    }
    prefix_bit = prefix_bit >> 1;
}
```

注意到上述循环共 32 次，这是因为前缀树最多也就 32 层。但有一种情况可以 32 次都匹配到子节点，即某一 ip 的特殊路由，前缀长度为 32。因此在跳出循环后，当前节点一定是位于 32 层的匹配节点，我们直接返回当前节

点的 port_id。为了确保正确，我们可以再检查一下其节点类型，如果不是匹配节点则报错。在实际运行 forwarding-table.txt 数据集时，没有任何报错，因此可以确定我们的分析是正确的。

```
// case: prefix_len == 32
if (node_now->type == MATCH) {
    return node_now->port_id;
}
else {
    printf("error: 1\n");
    return -1;
}
```

3、数据集测试和结果验证

首先我们需要验证这个最基本前缀树的正确性，使用宏定义 CHECK，当其有效时在查询过程中进行对比和打印信息。基于 forwarding-table.txt 数据集每个条目的 ip 进行查询，将查询结果 port 与该条目的 port 进行对比，一致则打印 true，否则打印 false。

```
#ifdef CHECK
FILE *fp_out = fopen("lookup_result.txt", "w");
#endif

gettimeofday(&start, NULL);
for (int i=0; i<total_entry_num; i++) {
    Tree_Lookup(data_entry[i].ipv4, root);
    #ifdef CHECK
    int port = Tree_Lookup(data_entry[i].ipv4, root);
    char ipv4_str[20];
    int_to_ipv4(data_entry[i].ipv4, ipv4_str);
    fprintf(fp_out, "%s %d %d %d ", ipv4_str, data_entry[i].prefix_len, data_entry[i].port_id, port);
    if (port == data_entry[i].port_id)
        fprintf(fp_out, "true\n");
    else
        fprintf(fp_out, "false\n");
    #endif
}
gettimeofday(&end, NULL);
time_use = (end.tv_sec - start.tv_sec)*1000000 + (end.tv_usec - start.tv_usec);
```

这个简单的对比测试并不完善，因为根据 CIDR 机制，查询结果不一定等于该条目中的端口。

如果存在如下两个条目：128.0.0.0, 1, 3;　　128.0.0.0, 2, 4;

先后两次查询 128.0.0.0，其结果应该都返回 4。这时第一个条目会打印“false”，但实际这是正确的行为。

首先对上述相同 ip 情况进行测试，结果如下。每一行是一个条目的查询结果，依次为：ip、前缀长度、给定端口号、查询结果端口号、对比结果。

可以看到在第一个 128.0.0.0 条目处，查询结果为 5，打印为“false”，这与预期情况一致，这说明该前缀树可以正常处理该情况。

```
0.0.0.0 1 1 1 true
128.0.0.0 1 2 5 false
128.0.0.0 2 5 5 true
192.0.0.0 2 6 6 true
240.0.0.0 4 10 10 true
```

另一个需要注意的是重叠冲突的情况，上图中 128.0.0.0,1,2 条目实际涵盖 128.0.0.0,2,5 和 192.0.0.0,2,6 条目，但根据最长匹配原则，查询 128.0.0.0 和 192.0.0.0 时，结果以后两个条目为准。

我们从上图可以看到，确实查询结果以最长为准，这说明该前缀树也可以正常处理冲突情况。

至此我们验证了该基础前缀树 IP 路由查找算法的正确性，之后的高级算法就可以以该基础算法的结果为准，看查找结果是否正确。

（二）高级 IP 前缀查找方案——Leaf Pushing

1、Leaf Pushing 思路

我们回顾一下基础前缀树查找的流程，从根节点逐位匹配，直到无子节点。查看当前节点的类型，不是匹配节点则向后回溯，找到最近的匹配节点。

在实际查找中的问题为，回溯情况非常普遍，这将带来大量重复访存。于是我们想到能不能避免回溯，在最后匹配停止的节点上直接获得其上方最近一个匹配节点的值。这就是 Leaf Pushing 的出发点，将上层匹配节点的值推到下方的叶子节点，以避免回溯。

2、Leaf Pushing 实现与改进

Leaf Pushing 的实现方法为，从根节点开始，如果其子节点是内部节点，将该节点的值推给子节点；如果子节点已经是匹配节点，不用更改。如此递归进行，就可以将匹配节点的值推向所有内部节点。

然后我们可以考虑改进一下树节点数据结构。既然已经不用回溯，那么 parent 指针自然也不必要了。最后所有节点的 port 域都为有效值，也就不用 type 来区分节点类型。网络号和掩码长度，其实隐含在树结构中，可以在逐位匹配过程中得到。

最终树节点数据结构改进为：

```c
typedef struct node {
    int port_id;
    struct node* left;
    struct node* right;
} TreeNode;
```

3、Leaf Pushing 关键函数

Tree_Leaf_Pushing 负责实现叶推，在构建前缀树完成后，调用该函数来实现匹配节点叶推和化简树。

```
void Tree_Leaf_Pushing(TreeNode *node_now, int parent_port){
    // if it is a internal node, inherits the port from parent node
    if (node_now->port_id == -1) {
        node_now->port_id = parent_port;
    }

    // push port to leaf node
    if (node_now->left) {
        Tree_Leaf_Pushing(node_now->left, node_now->port_id);
    }
    if (node_now->right) {
        Tree_Leaf_Pushing(node_now->right, node_now->port_id);
    }
}
```

Tree_Lookup 查找函数，去掉了回溯，在逐位匹配停止后直接返回当前节点的 port_id。

```
int Tree_Lookup(int ipv4, TreeNode *root){
    TreeNode *node_now = root;
    uint32 prefix_bit = 0x80000000;
    int i, port;
    for (i = 1; i <= 32; i++) {
        if ((ipv4 & prefix_bit) == 0) {
            if (node_now->left == NULL) {
                return node_now->port_id;
            }
            node_now = node_now->left;
        }
        else {
            if (node_now->right == NULL) {
                return node_now->port_id;
            }
            node_now = node_now->right;
        }
        prefix_bit = prefix_bit >> 1;
    }

    // case: prefix_len == 32
    return node_now->port_id;
}
```

4、正确性验证

以基础前缀树查找结果为基准，检查 Leaf Pushing 的 IP 前缀查找是否正确。

```
joker@joker-linux:/mnt/hgfs/share/10-lookup$ diff ./advanced/lookup_result.txt
./basic/lookup_result.txt
joker@joker-linux:/mnt/hgfs/share/10-lookup$
```

通过 diff 发现查找结果与基础前缀树结果一致，验证了 Leaf Pushing 的正确性。

（三）性能评估

1、基础前缀树测试结果

测试结果如下：

```
joker@joker-linux:/mnt/hgfs/share/10-lookup/basic$ ./basic_lookup
Total Entry: 697882
Total Memory: 65863400 B
Total Time: 62279.000000 us
Time per lookup: 89.240015 ns
```

内存消耗为 65863400 B = 62.8 MB，平均查找时间为 89.24 ns。

2、Leaf Pushing 测试结果

测试结果如下：

```
joker@joker-linux:/mnt/hgfs/share/10-lookup/advanced$ ./advanced_lookup
Total Entry: 697882
Total Memory: 39518040 B
Total Time: 62039.000000 us
Time per lookup: 88.896117 ns
```

内存消耗为 39518040 B = 37.7 MB，平均查找时间为 88.896 ns。

3、对比分析

两种方法的测试结果对比可知，Leaf Pushing 的内存开销减少约 40%，这是由于避免回溯而简化了树节点数据结构。然而平均查找时间的减少并不明显，这可能是因为在路由表项非常多的情况下，每条路径上的匹配节点之间的距离不会太大，每次回溯层数不多，因此避免回溯消除的访存不多。

四、实验总结

通过本次实验，我了解了实际使用的 IP 路由查找算法，也实现了最基础的前缀树查找方法。之后通过调研也了解到一些高级的查找算法，例如 Leaf Pushing、Tree Bitmap 等等，加深了我对 IP 路由查找的理解。