高效ip路由查找实验报告

张翔雨 2018K8009929035

一、实验题目：高效路由ip查找实验

二、实验内容

- 实现最基本的前缀树查找
- 调研并实现某种IP前缀查找方案
- 基于forwarding-table.txt数据集(Network, Prefix Length, Port)
  - 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
  - 对比基本前缀树和所实现IP前缀查找的性能

三、实验过程

1.完成最基本的前缀树查找。

（1）实现数据处理

调用strtok函数将输入行分成对应的部分。再使用sscanf函数对字符串格式化输出，按每8位分配。得到一个info结构体，定义如下：

```
typedef struct line{
    unsigned int ip;
    int prefix_len;
    int port;
} Info;
```

数据处理代码如下：

```
while(fgets(line, MAXLINE, fd) != NULL)
    {
        Info* new_info = (Info*)malloc(sizeof(Info));

        char *ip_char = strtok(line, " ");
        new_info->prefix_len = atoi(strtok(NULL," "));
        new_info->port = atoi(strtok(NULL," "));
```

```
        sscanf(ip_char,"%u.%u.%u.%u",&ip_temp[0],&ip_temp[1],&ip_temp[2],&ip_temp[3]);

        input[input_num] = ip_temp[0];
        input[input_num] = (input[input_num] << 8) + ip_temp[1];
        input[input_num] = (input[input_num] << 8) + ip_temp[2];
        input[input_num] = (input[input_num] << 8) + ip_temp[3];

        new_info->ip = input[input_num];
        insert_info(new_info);
        insert_info_2bit(new_info);
        input_num++;
    }
```

（2）实现1bit前缀树的构造

1bit前缀树节点数据结构如下：

```
typedef struct Node {
    int port;
    struct Node* left;
    struct Node * right;
} PrefixTree;
```

只有左右孩子和对应的端口号，在前面的数据处理部分中可以看到每读到一行的输入就插入一行的数据，插入1bit前缀树的函数实现如下

```
int insert_info (Info* new_info)
{
    PrefixTree * ptr = root;
    unsigned int mask = 0x80000000;

    int insert_num = 0;
    for (int i = 0; i < new_info->prefix_len; i++)
    {
        if ((new_info->ip & mask) == 0)
        {
            if (ptr->left == NULL)
            {
                PrefixTree *tmp = (PrefixTree *) malloc(sizeof(PrefixTree));
                space_cost += sizeof(PrefixTree);
                tmp->port = ptr->port;
                ptr->left = tmp;
            }
            ptr = ptr->left;
        }
        else
        {
            if (ptr->right == NULL)
```

```
            {
                PrefixTree *tmp = (PrefixTree *) malloc(sizeof(PrefixTree));
                space_cost += sizeof(PrefixTree);
                tmp->port = ptr->port;
                ptr->right = tmp;
            }
            ptr = ptr->right;
        }
        mask = mask >> 1;
    }
    ptr->port = new_info->port;
    return 1;
}
```

基本实现思路是对输入的ip地址逐位与1进行与运算来判断是1或0,结果是0则进入当前节点的左节点，如果左节点不存在，则新分配一个新节点，端口号为父结点的端口号，结果是1时类似操作。当循环达到前缀长度时，将当前所在节点的端口号改为输入信息中的端口号，完成插入。当所有节点都插入后就构造了一颗完整的1bit前缀树。

（3）完成1bit前缀树的查找

1bit前缀树查找比较简单，只需要对输入ip的32位地址从高到低按位查找，为1进入右节点，为0进入左节点，直到找到空节点为止，代表找到了最长前缀，返回空节点的父结点端口号。

```
int lookup (int ip)
{
    PrefixTree * ptr = root;
    PrefixTree * ret = NULL;
    unsigned int mask = 0x80000000;
    int i;
    int port;
    for (i = 1; ptr; i++)
    {
        ret = ptr;
        if ((ip & mask) == 0)
            ptr = ptr->left;
        else
            ptr = ptr->right;
        mask = mask >> 1;
    }
    return ret->port;
}
```

2.调研优化ip路由查找算法

本次实验实现的是2bit前缀树查找方法，在1bit前缀树的基础上进行优化。

（1）实现2bit前缀树的构造

2bit前缀树节点数据结构如下：

```
typedef struct Node2{
    int port;
    struct Node2* son00;
    struct Node2* son01;
    struct Node2* son10;
    struct Node2* son11;
} PrefixTree2;
```

与1bit前缀树相似，不同的是两个孩子节点增加到了4个，分别对应1bit前缀树中左左、左右、右左、右右的情况。

同样的在读取到一行数据后将其插入到2bit前缀树中.

```
int insert_info_2bit (Info* new_info) {
    PrefixTree2 * ptr = root2;
    unsigned int mask = 0x80000000;
    unsigned int mask_next = 0x40000000;

    int insert_num = 0;
    for (int i = 0; i < new_info->prefix_len-1; i+=2)
    {
        if ((new_info->ip & mask) == 0)
        {
            if((new_info->ip & mask_next) == 0)
            {
                if(!ptr->son00)
                {
                    PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                    space_cost_2bit += sizeof(PrefixTree2);
                    tmp->port = ptr->port;
                    ptr->son00 = tmp;
                }
                ptr = ptr->son00;
            }
            else
            {
                if(!ptr->son01)
                {
                    PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                    space_cost_2bit += sizeof(PrefixTree2);
```

```
                tmp->port = ptr->port;
                ptr->son01 = tmp;
            }
            ptr = ptr->son01;
        }
    }
    else
    {
        if((new_info->ip & mask_next) == 0)
        {
            if(!ptr->son10)
            {
                PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                space_cost_2bit += sizeof(PrefixTree2);
                tmp->port = ptr->port;
                ptr->son10 = tmp;
            }
            ptr = ptr->son10;
        }
        else
        {
            if(!ptr->son11)
            {
                PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                space_cost_2bit += sizeof(PrefixTree2);
                tmp->port = ptr->port;
                ptr->son11 = tmp;
            }
            ptr = ptr->son11;
        }
    }
    mask = mask >> 2;
    mask_next = mask_next >> 2;
}
if(new_info->prefix_len%2)
{
    if((new_info->ip & mask) == 0)
    {
        if(!ptr->son00)
        {
            PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
            space_cost_2bit += sizeof(PrefixTree2);
            tmp->port = new_info->port;
            ptr->son00 = tmp;
        }
        if (!ptr->son01)
        {
            PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
            space_cost_2bit += sizeof(PrefixTree2);
            tmp->port = new_info->port;
            ptr->son01 = tmp;
        }
```

```
        }
        else
        {
            if(!ptr->son10)
            {
                PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                space_cost_2bit += sizeof(PrefixTree2);
                tmp->port = new_info->port;
                ptr->son10 = tmp;
            }
            if (!ptr->son11)
            {
                PrefixTree2 *tmp = (PrefixTree2 *) malloc(sizeof(PrefixTree2));
                space_cost_2bit += sizeof(PrefixTree2);
                tmp->port = new_info->port;
                ptr->son11 = tmp;
            }
        }
    }
    else
        ptr->port = new_info->port;
    free(new_info);
    return 1;
}
```

2bit前缀树插入算法与1bit不同的是，逐位查看的时候不仅需要当前位的，还需要查看下一位的情况，因此需要两个掩码对ip做与运算，这样一次循环可以处理2bit的数据，因此循环的步长也需要改为2，循环内的比较操作与1bit类似，只不过将2个区间变为4个区间插入。

这里带来了新的问题，如果步长为2就会发现如果前缀长度为奇数，最后一个bit无法找到该插入的位置，卡在了两个节点之间，那么就需要特殊判断，将倒数第二个bit查找到的节点的对应空子节点的值端口号设置为这个信息的端口号，代表是一个新的最长前缀。如果是偶数直接修改查到的子节点的端口号即可。

（3）完成2bit前缀树的查找

```
int lookup_2bit (int ip)
{
    PrefixTree2 * ptr = root2;
    PrefixTree2 * ret = NULL;
    unsigned int mask = 0x80000000;
    unsigned int mask_next = 0x40000000;
    int i;
    for (i = 1; ptr; i +=2)
    {
        ret = ptr;
        if ((ip & mask) == 0)
        {
```

```
            if((ip & mask_next) == 0)
                ptr = ptr->son00;
            else
                ptr = ptr->son01;
        }
        else
        {
            if((ip & mask_next) == 0)
                ptr = ptr->son10;
            else
                ptr = ptr->son11;
        }
        mask = mask >> 2;
        mask_next = mask_next >> 2;
    }
    return ret->port;
}
```

2bit前缀树的查找相比较于1bit前缀树的查找所做的改变就是对2分查找变成了4个区间的查找，剩下的逻辑相似，同时步长值需要改为2.

四、实验结果

对原始数据进行去重，保留应该查找到的结果，生成一份正确结果的`answer.txt`，并将该文件作为测试正确性和测量时间的输入，进行测试，得到结果如下：

```
samuel@ubuntu:/mnt/hgfs/network-labs/Lab8/10-lookup$ ./lookup
*********************************Input**********************************
total mem 1bit: 39518040
total mem 2bit: 48109920
*********************************Check**********************************
input num: 666617
error num :0
*********************************Count**********************************
1bit tree time_use is 65.63 ns
2bit tree time_use is 49.96 ns
```

结果分析如下：

- error的判断逻辑为1bit的查找结果与2bit不同或者1bit的查找结果与正确结果不同，可以看到error_num为0，证明程序正确。
- 从空间消耗来看，2bit前缀树消耗了比1bit前缀树更多的空间开销，也是意料中的，因为2bit前缀树的节点一旦生成就带有指向四个子节点的指针，空间开销明显大于1bit的节点。空间开销增加了约21.74%
- 从时间消耗来看，2bit前缀树的时间消耗明显少于1bit前缀树，减少了约23.88%。
- 总的来看，2bit前缀树是典型的拿空间换时间，通过使用更大的空间开销来换取时间性能上的提升。

在编译时可以尝试编译优化，观察对实验结果的影响，发现O2及以上的优化会导致测得时间为0，O1优化结果如下;

```
samuel@ubuntu:/mnt/hgfs/network-labs/Lab8/10-lookup$ gcc main.c -o lookup -O1
samuel@ubuntu:/mnt/hgfs/network-labs/Lab8/10-lookup$ ./lookup
*****************************Input*********************************
total mem 1bit: 39518040
total mem 2bit: 48109920
*****************************Check*********************************
input num: 666617
error num :0
*****************************Count*********************************
1bit tree time_use is 38.99 ns
2bit tree time_use is 37.41 ns
```

可以看到对性能的优化效果是比较明显的。

五、实验总结

本次实验不再沿用之前的框架，是独立实现一种功能，自由性比较高，但代码量依然不小。树的使用在数据结构结课后很久没有再遇到，本次实验刚开始写起来还十分生疏，许多debug的技巧也忘记了，通过本次实验回忆了这部分的知识。同时调研和自己实现部分也加深了对路由查找的理解。