计算机网络研讨课实验报告

冯吕 2015K8009929049

2018 年 5 月 12 日

实验题目

高效 IP 路由查找实验

实验内容

本次实验需要实现路由查找中的 IP 最长前缀查找。
实验内容分为两部分：

- 实现最基本的单比特匹配的前缀树查找；
- 实现多比特的前缀树查找以及优化：叶推、压缩向量以及压缩指针。

然后，基于数据集 $forwarding-table.txt$，来进行测试，并比对两种不同方法的性能开销。

实验流程

单比特前缀树

在本次实验中，首先要实现最基本的前缀树查找，实现该查找的类定义如下：

```
class PTree{
        public :
                u32 SubNet;
                u8 PreLen;
                u8 Port;
                PTree(){
                        SubNet = 0;
                        PreLen = 0;
                        Port = 0;
                        Child[0] = Child[1] = NULL;
                };
                u32 StrToIp(const char *s);
                virtual bool ConstructTree();
                bool Insert(PTree *Node);
                bool Pleafpush(PTree *Node);
                PTree *Search(u32 IP);
                virtual bool DestructTree();
```

```
18                  virtual ~PTree(){
19                  };
20                  void dumpNode();
21          private :
22                  PTree *Child[2];
23  };
```

在该类的成员中，*SubNet* 存储子网，*PreLen* 存储前缀长，*Port* 存储对应的端口，以及两个指向子树的指针数组。构建树时将子网转换成一个 32 位无符号整数。因此，定义了一个将 *str* 转化为 32 位无符号数的方法。构建树时，将数据集中的每一条记录依次插入到树中，构建树的过程实际上也是一个查找的过程。查找时，则将查找 *IP* 和前缀树进行前缀匹配搜索。

dumpNode 方法将查找结果以可读形式输出到标准输出。另外，当程序退出前，需要将前缀树释放 (*DestructTree*())。

多比特前缀树

多比特前缀树的定义如下：

```
1   class MulPTree : public PTree{
2           public :
3                   MulPTree *mChild[4];
4                   MulPTree(){
5                           SubNet = 0;
6                           PreLen = 0;
7                           Port = 0;
8                           mChild[0] = mChild[1] = mChild[2] = mChild[3] = NULL;
9                   };
10                  virtual bool ConstructTree();
11                  bool Insert(MulPTree *Node);
12                  MulPTree *Search(u32 IP);
13                  bool MleafPush(MulPTree *Node);
14                  bool Equal(PTree *Node){
15                          return Node->SubNet == SubNet
16                          && Node->PreLen == PreLen
17                          && Node->Port == Port;
18                  }
19                  bool Equal(Leaf *leaf){
20                          return SubNet == leaf->SubNet
21                          && PreLen == leaf->PreLen
22                          && Port == leaf->Port;
23                  };
24                  virtual bool isLeaf(){
25                          return !mChild[0] && !mChild[1] && !mChild[2]
26                          && !mChild[3];
27                  }
28                  virtual bool DestructTree();
```

```
29                  virtual ~MulPTree(){
30                  };
31 };
```

多比特前缀树继承单比特前缀树，它的构建和查找方法和前者基本一样。它的指向子树的指针数组大小变为 4，另外，增加了判断两个判断查找结果是否相等的方法。

叶推

叶推时，需要把所有的包含匹配前缀的中间节点下推到叶子节点。通过递归能够很方便的实现。

下面是多比特前缀树的叶推方法，单比特类似：

```
1  bool MulPTree::MleafPush(MulPTree *Node){
2          if (mChild[0] == NULL && mChild[1] == NULL
3          && mChild[2] == NULL && mChild[3] == NULL){
4                  return true;
5          }
6          if (SubNet != 0){
7                  Node->SubNet = SubNet;
8                  Node->Port = Port;
9                  Node->PreLen = PreLen;
10         }
11         if (mChild[0]&&mChild[1]&&mChild[2]&&mChild[3]){
12                 for (u8 i = 0; i != 4; ++i){
13                         mChild[i]->MleafPush(Node);
14                 }
15         }
16         else {
17                 MulPTree *NewNode = new MulPTree();
18                 NewNode->SubNet = Node->SubNet;
19                 NewNode->PreLen = Node->PreLen;
20                 NewNode->Port = Node->Port;
21                 for (u8 i = 0; i != 4; ++i){
22                         if (mChild[i] == NULL)
23                           mChild[i] = Node;
24                         else
25                           mChild[i]->MleafPush(NewNode);
26                 }
27         }
28         return true;
29 }
```

压缩指针和压缩向量

压缩指针和压缩向量是在叶推的基础上实现的，叶推之后，除了叶子节点之外，所有的中间节点的指向孩子的指针均不为空，因此，可以把中间节点和叶子节点分开，中间节点存储子孩子的标记：是叶子还

是子树，以及两个指向叶子向量和子树向量的指针，而叶子节点只需要存储数据即可。

下面是中间节点的定义。

```
class CVTree{
        public :
                bool NotLeaf[4];
                Leaf *LeafVec;
                CVTree *ChildVec;
                CVTree(){
                        NotLeaf[0]=NotLeaf[1]=NotLeaf[2]=NotLeaf[3]=0;
                        LeafVec = NULL;
                        ChildVec = NULL;
                }
                bool ConstructCVTree(MulPTree *Tree);
                Leaf *Search(u32 IP);
                u8 Count(u8 loc);
                bool DestructCVTree();
                virtual ~CVTree(){
                        if (LeafVec){
                                delete[] LeafVec;
                        }
                };
};
```

ConstructCVTree 方法构建压缩指针和向量后的树，它通过前序遍历一棵完成叶推后的前缀树来进行构建，压缩指针和压缩向量可以同时实现：

```
bool CVTree::ConstructCVTree(MulPTree *Tree){
        u8 haveChild = 0;
        u8 haveLeaf = 0;
        for(u8 i = 0; i != 4; ++i){
                if (Tree->mChild[i] && !Tree->mChild[i]->isLeaf()){
                        NotLeaf[i] = true;
                        ++haveChild;
                }
                if (Tree->mChild[i] && Tree->mChild[i]->isLeaf()){
                        ++haveLeaf;
                }
        }
        if (haveLeaf + haveChild != 4){
                return false;
        }
        if (haveLeaf){
                LeafVec = new Leaf[haveLeaf];
                for (u8 i = 0, j = 0; i != 4 && j != haveLeaf; ++i){
                        if (Tree->mChild[i]&& Tree->mChild[i]->isLeaf()){
```

```
20                                  LeafVec[j].SubNet = Tree->mChild[i]->SubNet;
21                                  LeafVec[j].PreLen = Tree->mChild[i]->PreLen;
22                                  LeafVec[j].Port = Tree->mChild[i]->Port;
23                                  ++j;
24                          }
25                  }
26          }
27          if (haveChild){
28                  ChildVec = new CVTree[haveChild];
29                  for (u8 i = 0, j = 0; i != 4 && j != haveChild; ++i){
30                          if (Tree->mChild[i] && !Tree->mChild[i]->isLeaf()){
31                                  ChildVec[j].ConstructCVTree(Tree->mChild[i]);
32                                  ++j;
33                          }
34                  }
35          }
36          return true;
37  }
```

实验结果

该程序运行时，输入为以点分隔的合法 IP，由于没有对输入的合法性进入检查，因此，如果输入不合法会导致程序 $crash$。

```
17:45 fenglv@segmentfault:09-lookup $ ./tree
Constructing Tree...

Input search IP: 1.0.4.3

Search Result:
Subnet: 1.0.4.0
Prefix Len: 24
Port: 3
Search Time of Single Bit Prefix Tree: 0.000010
Search Time of Double Bit Prefix Tree: 0.000006
Search Time of CV(Vector Compression) Prefix Tree: 0.000006

Input search IP: 100.42.225.32

Search Result:
Subnet: 100.42.225.0
Prefix Len: 24
Port: 0
Search Time of Single Bit Prefix Tree: 0.000012
Search Time of Double Bit Prefix Tree: 0.000008
Search Time of CV(Vector Compression) Prefix Tree: 0.000008

Input search IP: 99.49.48.0

Search Result:
Subnet: 99.49.48.0
Prefix Len: 22
Port: 2
Search Time of Single Bit Prefix Tree: 0.000012
Search Time of Double Bit Prefix Tree: 0.000007
Search Time of CV(Vector Compression) Prefix Tree: 0.000006

Input search IP:

Test End. All Test Cases Passed.
```

图 1: IP 查找

从性能上来看，单比特的查找时间大概是双比特的两倍，因为查找过程中需要访问的节点是它的两倍。

结果分析

本次实验原本以为会比较容易，但最后实现起来发现遇到了很多的 bug，调试时间远远超过写代码的时间，最后还导致作业没有能够按时提交实验，不过说到底还是自己的代码功底需要提升。