Prefix Trees and Longest Prefix Matching

81356 Duarte Miguel Ferreira Dias
81701 Francisco Pereira Rosa Correia Pombal
81759 João Santiago Morais Valente

Friday 13th October, 2017

1 Files and structure

Required features is implemented in tree.c and tree.h. Extra functionality is in tree2.c and tree2.h. Supporting data structures for input file reading are in table.c and table.h. Helper functions for user interaction are in functions.c and functions.h.

2 Main Functions (required features)

2.1 PrefixTree()

Receives a prefix table as an argument and returns the equivalent prefix tree. The file with the table was alreay read into a prefix_table using a support function. A prefix_table data structure is simply a linked list with a prefix string and a next hop as payload (see table.c and table.h for implementation details)

```
PrefixTree(prefix_table)
    create prefix_tree tree_root
    for(each table_entry of prefix_table){
        if(prefix = "e"){
            assign nextHop to tree_root
        }else{
            tree_aux := tree_root
            for(each prefix[i]){
                if(prefix[i] = 0)
                    tree_aux := tree_aux->zero
                if(prefix[i] = 1)
                    tree_aux := tree_aux->one
            } // end for
        } //end of else
            tree_aux->nextHop := nextHop
    } //end of for
return tree_root
```

2.2 PrintTable()

Receives a prefix tree and an empty address, prints the prefix tree in the terminal. This function is recursive in order to travel through all the prefix tree nodes.

```
PrintTable(prefix_tree, empty_address)
    if(tree_node->nextHop != -1) //-1 is equivalent to having no nextHop
        printf(prefix, nextHop)
    if(tree_node->zero != NULL)
        PrintTable(tree_node->zero, address + '0')
    if(tree_node->one != NULL)
        PrintTable(tree_node->one, address + '1')
```

2.3 LookUp()

Receives a prefix tree and an address, returns the corresponding address' nextHop. An aux pointer travels to the corresponding prefix of the address, it then sees if there is a nextHop for that prefix or not printing the corresponding nextHop in case it exists.

```
LookUp(prefix_tree, address)
    for(each address[i])
        if(address[i] = '0')
            tree_aux = tree_aux->zero
        if(address[i] = '1')
            tree_aux = tree_aux->one
        if(tree_aux = NULL){
            if(nextHop != -1)  //-1 is equivalent to that node having no nextHop
                return nextHop
            else
                printf(No nextHop found associated to this prefix)
            }
        if(tree_aux->nextHop != -1)  //-1 is equivalent to that node having no
            nextHop
            nextHop = tree_aux->nextHop
    }
```

2.4 InsertPrefix()

Receives a prefix tree, an address and a nextHop value, returns a prefix tree with the new prefix included. This function is similar to the previous one (LookUp()), but in the end, instead of returning the value of the nextHop, it changes the value of the current node's nextHop to the desired one

2.5 DeletePrefix()

Receives a prefix tree and the address of the prefix to be deleted, returns a prefix tree without the prefix specified by the address.

```
DeletePrefix(prefix tree, address)
    for(each address[i]){
        if(address[i] = '0'){
            Delete_direction = LEFT
            tree_aux = tree_aux->zero
        }
        if(address[i] = '1'){
            delete_direction = RIGHT
            tree_aux = tree_aux->one
        }
    }
    if(node is not a leaf){
        tree_aux->nextHop = NO_HOP
    }
    else{
        if(delete_direction = LEFT){
            freeNode(node->left)
        }
        if(delete_direction = RIGHT){
            freeNode(node->one)
        }
    }
```

3 Extra Functions

3.1 BinaryToTwoBit()

Receives a binary prefix tree and converts it to a two bit prefix tree which is returned. This function is composed by two sub-functions: BinaryToTwoBit_recursive() and TreeNode_2_buildNode().

BinaryToTwoBit_recursive() receives a binary prefix tree, a two bit prefix tree, a nextHop value and an empty address as an input, it then travels throught all of tree nodes recursively, outputting the corresponding's node address to TreeNode_2_buildNode() in case the node has a prefix.

TreeNode_2_buildNode() receives a two bit prefix tree, an address and a nextHop value as an input, it then creates a new two bit tree node associated to the received address and all the necessary nodes that connect it to the head of the tree.

```
BinaryToTwoBit(binary tree){
    create new TwoBitTreeNode and empty address
    BinaryToTwoBit_recursive(binary tree, two bit prefix tree, empty address)
    return two bit prefix tree
}

BinaryToTwoBit_recursive(treeNode, 2BitTreeNode, address){
    if(treeNode->zero != NULL){
        address = address + '0'
    }
    BinaryToTwoBit_recursive(treeNode->zero, 2BitTreeNode, address)
    if(treeNode->one != NULL){
        address = address + '1'
    }
    BinaryToTwoBit_recursive(treeNode->one, 2BitTreeNode, address)

    // In the real code a lot of edge cases are covered in detail, but to here we
        need to simplify the code after the recursive calls in order to keep the
        report short:
    figure out the new_adress, depending on parity of leve;, number of children...,
         then:
    TreeNode_2_buildNode(2BitTreeNode, new_address, treeNode->nextHop)
}

TreeNode_2_buildNode(2BitTreeNode, 2BitAddress, nextHop){
    for(number of times needed to reach the required 2BitAddress){
        if(next two bits of 2BitAddress are "00"){
            if(2BitTreeNode->zero = NULL){
                2BitTreeNode->zero = newTreeNode_2()
            }
            2BitTreeNode = 2BitTreeNode->zero
        }
        if(next two bits of 2BitAddress are "00"){
            if(2BitTreeNode->one = NULL){
                2BitTreeNode->one = newTreeNode_2()
            }
            2BitTreeNode = 2BitTreeNode->one
        }
        if(next two bits of 2BitAddress are "10"){
            if(2BitTreeNode->two = NULL){
                2BitTreeNode->two = newTreeNode_2()
            }
            2BitTreeNode = 2BitTreeNode->two
        }
        if(next two bits of 2BitAddress are "11"){
            if(2BitTreeNode->three = NULL){
                2BitTreeNode->three = newTreeNode_2()
            }
            2BitTreeNode = 2BitTreeNode->three
        }
    }

    set nextHop of 2BitTreeNode
}
```

3.2 PrintTableEven()

This function is similar to PrintTable(), except there are four recursion paths instead of two (one for concatenating "00", one for concatenating "01", one for concatenating "10" and one for concatenating "11" with the empty address).