Prefix Trees and Longest Prefix Matching

Nuno Venturinha 67682
nuno.venturinha.f@tecnico.ulisboa.pt

João Borrego 78990
joao.borrego@tecnico.ulisboa.pt

I. Introduction

This project focuses on the implementation and analysis of basic functions involving prefix trees for fast binary address lookup. Our report describes the main aspects of the implemented algorithms and includes a pseudocode description for those that are non-trivial. It concludes with a brief discussion on the topic with examples of alternative methods.

II. Algorithms

Our project focuses on a given set of functions that operate on the binary tree representation of a prefix table.

A. Binary Prefix Tree

1) Prefix insertion: The insertion of a prefix on a binary tree is detailed in Algorithm 1.

Algorithm 1: Insert prefix in binary prefix tree

```
input : root 1-bit prefix tree root
        prefix Prefix to insert
        next_hop Next hop for the given prefix
output: bin_tree 1-bit prefix tree

Function insert(root, prefix, next_hop)
    curr_node ← root;
    if curr_node is not allocated then
        curr_node ← newNode(EMPTY);
    end
    if prefix has ended then
        cur_node.value ← next_hop;
    else if first bit of prefix is '0' then
        curr_node.left ← insert(curr_node.left, prefix+1,
          next_hop);
    else if first bit of prefix is '1' then
        curr_node.right ← insert(curr_node.right,
          prefix+1, next_hop);
    end
    return curr_node
```

This seems to be an intuitive and efficient approach. Our solution introduces support nodes, i.e., nodes that do not correspond to a valid prefix but are still necessary in order to reach more specific prefixes. These nodes hold no next-hop value and are instead flagged as empty.

The recursion is performed as many times as the number of bits[1] in the prefix. For this reason, we can say that the algorithm has a temporal complexity of $\mathcal{O}(b)$, b number of bits in the given prefix.

A final remark should be added concerning an implementation detail: since prefixes are represented as strings (pointers), in order to advance a character it suffices to increment the respective pointer.

2) Creation from file: Creating a binary tree from a prefix table text file is as simple as reading each line and inserting the prefix and corresponding next-hop value in a tree using the prefix insertion function. This method makes no assumption on the data organisation and requires no preprocessing step. For this reason each insertion starts from the root of the tree.

3) Prefix deletion: The deletion of a prefix in the tree is done according to Algorithm 2.

Algorithm 2: Delete prefix from prefix tree

```
input : root 1-bit prefix tree root
        prefix Prefix to remove
output: action Action to perform on the given node

Function remove(root, prefix)
    if root is allocated then
        if first bit of prefix is '0' then
            if remove(root.left, prefix + 1) is equal to
              REMOVE then
                delete root.left;
                if root is EMPTY and root.right does not
                  exist then
                    return REMOVE
                else
                    return KEEP
                end
            end
        else if first bit of prefix is '1' then
            if remove(root.right, prefix + 1) is equal
              to REMOVE then
                delete root.right;
                if root is EMPTY and root.left does not
                  exist then
                    return REMOVE
                else
                    return KEEP
                end
            end
        else
            if root has no children then
                return REMOVE
            else
                root ← EMPTY;
                return KEEP
            end
        end
    end
    return NOT_FOUND
```

[1] Actually it is performed $b + 1$ times, but this is a minor detail and will be ignored from now on

Deleting a prefix is almost identical to the insertion up to the part of finding the desired node to operate on. However, one must take special care with freeing the memory properly while ensuring the rest of the tree is preserved. While this is mostly just a concern on the programming language chosen for the implementation (C++) it is still relevant.

Furthermore, there might be the case that an empty node is supporting the removed node, in which case it should be removed as well, provided it has no children. This may be performed several times, and is possible in our implementation due to a return value signalling whether the node should be kept or not. The prefix to be removed (if present) is also found with at most b recursion calls.

4) Print prefix table: Our implementation of the print prefix table function is a trivial Depth First Search and for this reason is not further detailed.

5) Lookup: The look up of a given address on the tree is done according to Algorithm 3:

Algorithm 3: Find prefix in prefix tree

```
input : root 1-bit binary prefix tree root
        address Address to find
output: next_hop Next hop for the given address

Function lookup(root, address)
    aux ← NOT_FOUND;
    if root is not allocated then
        aux ← END_REACHED;
    else
        if address has ended then
            if root is not EMPTY then
                aux ← root.value;
            else
                aux ← END_REACHED;
            end
        else if first bit of address is '0' then
            aux ← lookup(root.left, address+1);
        else if first bit of address is '1' then
            aux ← lookup(root.right, address+1);
        end
        if aux is equal to END_REACHED and root is
          not EMPTY then
            aux ← root.next_hop
        end
    end
    return aux;
```

This is the main application of this data structure and related algorithms. Having a prefix tree allows address lookups to be linear with the number of bits instead of having to search in 2^b space. In this sense, lookups have logarithmic complexity in the size of the domain. It should be noted that since we are dealing with prefixes, it is likely the case that the address is longer than a registered prefix. For this reason, if the recursion has not yet exhausted the bits in the address but reached a dead end, it should return the next-hop value of the current node. Once again, the traversal of the tree is done by means of recursion and as such it shares the complexities of the insertion and deletion algorithms.

B. Quad Prefix Tree

We were also asked to extend the idea of prefix trees to quad trees, which essentially allow checking two bits per level. This effectively cuts the depth of the tree in half. However we now have the constraint that each prefix has an even number of bits. In order to convert a regular binary prefix tree (also known as unibit trees) to a 2-bit prefix tree (2-bit expanded trees) one must ensure that they produce the same next-hop value for a given address.

1) Binary to 2 bit conversion: Our first naive implementation is shown in algorithm 4.

Algorithm 4: Convert 1-bit prefix tree to 2-bit

```
input : bin_root 1-bit prefix tree root
        quad_root 2-bit prefix tree root
        prefix Prefix
        depth Depth
output: quad_tree 2-bit prefix tree

Function convert(bin_root, quad_root, prefix, depth)
    if bin_root is allocated then
        if next_hop is not EMPTY then
            if depth is even then
                insert_quad_node(quad_tree, prefix,
                  next_hop);
            else
                prefix[depth] ← '0';
                insert_quad_node(quad_tree, prefix,
                  next_hop);
                prefix[depth] ← '1';
                insert_quad_node(quad_tree, prefix,
                  next_hop);
            end
        end
        prefix[depth] ← '0';
        convert(bin_tree.left, quad_tree, prefix, depth+1);
        prefix[depth] ← '1';
        convert(bin_tree.right, quad_tree, prefix, depth+1);
    end
    /* The quad tree is altered by
       reference, so there is no return
       value                              */
```

The method correctly produces an equivalent prefix tree. The main idea here is that the algorithm behaves differently if it is operating on a level with even or odd depth. If it is at an odd level it inserts the current's node next-hop value in both expansions of the given prefix (With a 0 or a 1 added at the end to make it even). For instance, if at node with corresponding prefix '000' and next-hop 5 the algorithm will insert the prefixes '0000' and '0001' with next-hop value 5 in the quad tree. In nodes with an even depth it is only needed to update the value of the next-hop in the corresponding node in the quad tree or insert it of needed.

However this approach is inefficient, in the sense that the insertion of each prefix in the 2-bit prefix tree is performed at its root node, which results in a complexity of $\Theta(n \log n)$. Each node of the original tree has to be traversed, which performs a lower bound of n on complexity. Since each recursion call effectively corresponds to restricting to a sub-tree, it seems intuitive that we can leverage this property in order to insert the prefix without returning to the root. By changing the focus to the quad tree itself, visiting two levels of the binary tree at once and performing four recursion calls per node on the quad tree, we effectively tackle our initial problem. The resulting algorithm is presented in Algorithm 5.

While harder to understand, this algorithm provides $\mathcal{O}(n)$ complexity by keeping references to all the grandchildren of the current binary tree node and then performing the insertion on the new tree accordingly. Every insertion in the quad-tree is $\mathcal{O}(1)$ since the prefixes correspond to two bits that are directly inserted at the current node.

Algorithm 5: Convert 1-bit prefix tree to 2-bit

```
input :  bin_root 1-bit binary prefix tree root
         quad_root 2-bit quad prefix tree root
output:  quad_root Filled 2-bit quad prefix tree

Function convert (bin_root, quad_root)
    if bin_root exists then
        /* rXX are pointers to
           grandchildren of the binary
           root. vXX are the value of
           these nodes (NONE by default).
           */
        if bin_root.left exists then
            r00 ← bin_root.left.left;
            r01 ← bin_root.left.right;
            v00 ← bin_root.left.value;
            v01 ← bin_root.left.value;
        end
        if bin_root.right exists then
            r10 ← bin_root.right.left;
            r11 ← bin_root.right.right;
            v10 ← bin_root.right.value;
            v11 ← bin_root.right.value;
        end
        /* Only update the values if the
           nodes are not empty           */
        if r00 exists and r00 has a value then
            v00 ← r00.value;
        if r01 exists and r01 has a value then
            v01 ← r01.value;
        if r10 exists and r10 has a value then
            v10 ← r10.value;
        if r11 exists and r11 has a value then
            v11 ← r11.value;
        end
        if v00 has a value then
            insert(quad_root, '00', v00);
        if v01 has a value then
            insert(quad_root, '01', v01);
        if v10 has a value then
            insert(quad_root, '10', v10);
        if v11 has a value then
            insert(quad_root, '11', v11);
        end
        if quad_root exists then
            quad_root.child[b00] ←
              convert(r00, quad_root.child[b00]);
            quad_root.child[b01] ←
              convert(r01, quad_root.child[b01]);
            quad_root.child[b10] ←
              convert(r10, quad_root.child[b10]);
            quad_root.child[b11] ←
              convert(r11, quad_root.child[b11]);
        end
    end
    return quad_root
```

III. Discussion

This work consists in a fairly rudimentary approach to the topic of prefix trees and address lookup. The trees we implemented encode the knowledge of the prefix implicitly in the edges connecting the nodes. The choice of an edge in a given node corresponded to selecting 1 or 2 bits of the prefix even though the prefix itself is not present in any of the nodes. The nature of these trees resulted in the mentioned support nodes, which do not correspond to a valid prefix.

There are alternative approaches, namely involving combination of prefix trees and hash tables.

There are known techniques such as leaf-pushing and path compression. The former combines the principle of prefixes and leverages the use of pointers whereas the latter suppresses the support nodes effectively compressing the path into a single edge.

More complex approaches are present in recent literature.

[1] introduces the concept of prefix hash trees, which use distributed hashtables to achieve efficient lookups but also more robustness, since they are resilient to the failure of nodes. These are applied in the context of key-value pair indexing and lookups.

[2] obtained a static entropy-compressed representation of a forwarding table with optimal lookup. The authors' modified the prefix tree structure to incorporate entropy bounds an retain optimal lookup.

References

[1] Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. Prefix hash tree: An indexing data structure over distributed hash tables.

[2] Rétvári, G., Tapolcai, J., Kőrösi, A., Majdán, A., and Heszberger, Z. Compressing ip forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 111–122.