Algoritmia e Desempenho em Redes de Computadores

1º Mini Projeto - *Forwarding traffic in the Internet*

Bernardo Gomes, 75573

Tomás Falcato, 75876

11 de Outubro de 2015

1 Descrição do problema

Neste mini-projecto, pretende-se a implementação de diversas funções relacionadas com a transmissão de dados em pacotes consoante o seu destino.

Nesta fase, apenas se desenvolveram funções de conversão de *forwarding table* para *binary tree*, desta última para *binary2-tree*, adição de novos prefixos, remoção de prefixos e impressão de uma árvore binária no formato de *forwarding table*.

2 Função *ReadTable*

Nesta função o programa deverá ler de um ficheiro os prefixos de uma tabela e os correspondentes *next hops* e criar a árvore binária correspondente. Como tal, a função irá ler do ficheiro um prefixo de cada vez, adicionando-o à árvore com o *next hop* correspondente.

O pseudo-código da função será o seguinte:

```
rootnodecreation()
while there_are_lines_in_the_document do
    get_table_line()
    AddPrefix()
end while
     return
```

Desta forma, sendo o objectivo a inserção de todos os prefixos, será necessário fazer n vezes a chamada à função *AddPrefix*. A complexidade do algoritmo será assim O(n)O(get_table_line).

3 Função *AddPrefix*

Tal como descrito no enunciado, a função tem por objectivo a adição de um novo prefixo à árvore binária. Assumimos que esta função apenas é chamada quando a árvore binária do programa já foi previamente inicializada, ou seja, apenas é chamada dentro da função *ReadTable*, onde o *root node* é inicializado internamente, ou depois desta ser realizada.

Nesta função, testa-se se o prefixo a colocar na árvore é válido. No caso de ser, verifica se o prefixo é do *root* ou se é um endereço binário. Se for do *root*, copiam-se as informações directamente para a estrutura apontada pelo ponteiro. Se não for, ir-se-á percorrer os nós da árvore correspondentes (criando os não existentes no caminho), copiando os valores pretendidos na estrutura correspondente ao prefixo indicado.

No pior caso, o algoritmo implementado terá de percorrer a altura da árvore. A complexidade irá ser O(Comprimento do prefixo).

```
if adress_is_invalid then return  end if
    if adress_is_"*" then                ▷ root prefix
        root->next_hop := next_hop
        root->prefix := prefix
    else
        auxiliar_node := root_node
        while all_prefix_nodes_havent_been_visited_yet
do
            if bit_is_0 then
                if there_is_no_node then
                    node_creation()
                end if
                auxiliar := auxiliar->zero
            else                              ▷ bit is one
                if there_is_no_node then
                    node_creation()
                end if
                auxiliar := auxiliar->one
            end if
            auxiliar->next_hop := next_hop
        end while
    end if
```

4 Função *DeletePrefix*

Nesta função, assumimos os seguintes factos: o comando de *delete* do *root* deve ser considerado como inválido, e no caso de se apagar um prefixo que seja "filho único"o "pai"também deve ser removido.

Assim, no início da função, esta faz chamadas recursivas a ela própria até atingir o prefixo pretendido, retornando com uma mensagem de erro no caso de este não existir. A função *new_prefix* corresponde a retirar ao prefixo a apagar um bit, de forma a que o caminho a percorrer se inicie no bit onde a função se encontra de momento.Após apagar o nó correspondente, a função, ao retornar 1, "avisa"o nó de cima

("pai") de que o ponteiro para este será agora NULL. Este factor será bastante importante na medida em que a condição de *free* baseia-se no facto de os dois ponteiros para estruturas "filho"serem NULL.

A complexidade do algoritmo será O(comprimento do prefixo).

```
if prefix_corresponds_to_root then return -1 end if
 if havent_reached_the_end_of_the_prefix_bits then
    if bit = 0 then
        prefix := new_prefix()
        if node_exists then
            if Delete_prefix(base_node- > zero, prefix) = 1 then
                base_node- > zero := NULL
            end if
        else
            prefix_does_not_exist return -1
        end if

            if bit = 1 then
                prefix := new_prefix()
                if node_exists then
                    if Delete_prefix(base_node- > one, prefix) = 1 then
                        base_node- > one := NULL
                    end if
                else
                    prefix_does_not_exist return -1
                end if

                end if

                if node_has_no_children then
                    Free_node(base_node) return 1
                else
                    base_node- > next_hop := −1   ▷ cant free the node but erases next hop
                end if
                return 0
```

5 Função *PrintTable*

Neste caso, pretende-se a conversão de uma árvore binária numa *forwarding table*. Para tal, será necessário visitar todos os nós da árvore, verificando se têm *next hop* atribuído. Assim:

```
if has_next_hop then
    print_node_prefix_and_next_hop
end if
if has_"0"_child then
    PrintTable(zero_child)
end if
if has_"1"_child then
    PrintTable(one_child)
end if
    return
```

A árvore será assim percorrida em profundidade, da esquerda para a direita.

6 Função *TwoTree*

Com esta função, pretende-se a conversão da árvore obtida anteriormente pela função *ReadTable* para uma *binary2-tree*. Por definição, neste tipo de árvore, todos os nós que não são folhas têm exactamente dois "filhos". Desta forma, o algoritmo percorre recursivamente os nós da árvore. Se ao chegar a uma folha e a mesma não tiver um valor de *next hop* atribuído actualiza-o. Caso o nó não seja uma folha, verifica qual a ramificação em falta, cria-a e chama a rotina para cada ramo. A função tem como parâmetros de entrada um ponteiro para o topo da árvore/sub-árvore, bem como o valor de *next hop* que irá ser atribuído às folhas da ramificação em causa. A função *update_next_hop* tem como função verificar se o *next_hop* a utilizar na chamada recursiva é o mesmo no nó anterior ou um valor actualizado, no caso de o valor do nó visitado ser diferente.

```
if node_is_a_leaf then
    if has_no_next_hop then
        base_node- > next_hop := next_hop
    end if
else
    if has_no_"0"_child then
```

```
        prefix := new_node_prefix()
        create_zero_child(prefix)
    else
        if has_no_"1"_child then
            prefix := new_node_prefix()
            create_one_child(prefix)
        end if
    end if
    TwoTree(base_node->zero, update_next_hop())
    TwoTree(base_node->one, update_next_hop())
end if
    return
```

7 Função *AdressLookUp*

Com esta rotina, pretende-se retornar o *next_hop* do prefixo em causa. Considerando a ordem do enunciado, assume-se que a árvore a utilizar para obter a informação é do tipo *binary2-tree*. Como tal, toda a informação relativa aos *next_hops* estão contidas nas folhas da árvore. Assim, recebendo como argumentos de *input* o ponteiro para o topo da árvore e o prefixo desejado, o algoritmo em causa percorre os nós consoante os *bits* do prefixo especificado até que o ponteiro de um nó para o seu "filho"seja NULL. Quando tal acontecer significa que o nó para o qual o ponteiro auxiliar aponta é uma folha da árvore, cujo nó contém o *next_hop* mais específico para o prefixo dado no *input*.

```
if prefix_is_invalid then return -1
end if
auxiliar := root
while havent_reached_the_end_of_prefix_bits
do
    if bit = 0 then
        if has_no_"0"_child then return
auxiliar->next_hop
        end if
        auxiliar := auxiliar->zero
    else
        if has_no_"1"_child then return
auxiliar->next_hop
        end if
        auxiliar := auxiliar->one
    end if
end while
    return -1
```

8 Considerações finais