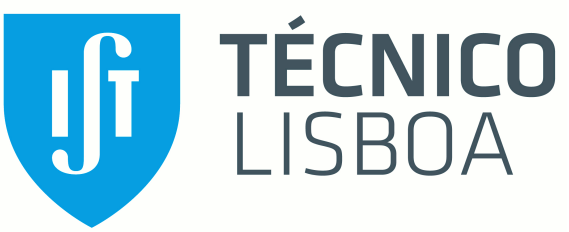

ALGORITMIA E DESEMPENHO EM REDES DE COMPUTADORES

MEEC

Prefix tables and longest match prefix rule

Grupo Nº 14

Autores:

Diogo Rodrigues (Nº 84030)
Pedro Moreira (Nº 85228)

Turno:

Quarta-Feira 12h30-14h00

18/10/2019

1 Introdução

Esta primeira parte do projeto é sobre tabelas de prefixos e a regra do prefixo de correspondência mais longa. Esta temática surge com a necessidade dos *routers* manterem uma tabela de encaminhamento de forma a saberem para ondem devem encaminhar os pacotes que lhe chegam consoante o seu endereço para um passo (*next-hop*) mais próximo do seu destino.

Associada a esta tabela existem várias operações possíveis (consulta, inserção e remoção), em que estas devem ser feitas o mais rapidamente possível, pelo que daí advém a necessidade de representar a tabela numa árvore (binária) de prefixos e aplicar algoritmos de forma a tornar estas operações eficientes, principalmente a de consulta. O tamanho destas tabelas de prefixos podem chegar a dimensões na ordem das centenas de milhares de entradas, pelo que é razoável pensar na possibilidade de encontrar uma forma de comprimir a árvore de prefixos mantendo o mesmo comportamente de reencaminhamento. Na secção 2 vai-se abordar com detalhe todos os algoritmos implementados para resolver cada um destes problemas.

2 Algoritmos usados

2.1 PrefixTree

A função *PrefixTree* lê uma tabela de prefixos de um ficheiro de texto com o seguinte formato: Prefixo seguido do *next-hop* dividos por um espaço em branco, e retorna a representação da árvore de prefixos binários dessa tabela.

O algoritmo implementado para a execução desta função lê cada linha do ficheiro de entrada com o formato predefinido. Assume-se que o prefixo lido com a letra 'e' será a saída *default* do *router* e a raíz da árvore de prefixos, contudo o algoritmo está preparado para o caso da não existência de uma saída *default*. Posteriormente, armazena o conteúdo lido dividindo o mesmo em prefixo e *next-hop*.

Para cada linha lida a função *InsertPrefix* é chamada e esta faz todo o processo de inserção como iremos ver detalhadamente na secção 2.4.

Complexidade: $\mathcal{O}(N)$, onde N = número total de *bits* na tabela de encaminhamento de entrada do programa.

2.2 PrintTable

A função *PrintTable* recebe como argumento a "raíz" da árvore de prefixos e exibe todos os possíveis *next-hops* de encaminhamento (com os seus prefixos associados), não devolvendo nenhum valor de retorno.

A função escreve para o *stdin*, de forma ordenada, uma tabela com duas colunas onde na primeira estará um prefixo e na segunda o *next-hop* (uma linha da tabela de encaminhamento). Para escrever os prefixos de forma ordenada recorreu-se a um FIFO, que é utilizado ao correr a árvore, a partir da raíz, da seguinte forma: antes de iniciar o ciclo principal, é inserido no FIFO o nó de raíz. Entrando no ciclo, em cada iteração é retirado um elemento do FIFO, que de imediato é escrito para o *stdin* (uma linha da tabela, com um prefixo e um *next-hop*). Depois são inseridos no FIFO os "filhos" do nó em questão (que acabou de ser escrito), e assim se segue até todos os nós serem escritos. Uma vez que na estrutura dos nós da árvore apenas consta o valor do *next-hop* e não o prefixo que representa, cada vez que um nó é inserido no FIFO, vai com ele associada uma *string*, que será igual à *string* do seu pai concatenada à frente com um '0' caso seja o filho da esquerda e com um '1' caso seja o filho da direita (a "raíz" é inserida com uma *string* vazia). Para entender melhor a linha principal de funcionamento da função, pode olhar-se para o pseudo-código apresentado a baixo.

Complexidade: $\mathcal{O}(N)$, onde N = número total de *bits* na tabela de encaminhamento (pior caso).

Algorithm 1 PrintTable(root)

```
put_in_FIFO (emptyString, root)
while FIFO not empty do
    (prefix_string, node) := get_from_FIFO()
    print(prefix_string, node->nexthop)
    if leftChild exists then put_in_FIFO(prefix_string + '0', node->leftChild)
    if rightChild exists then put_in_FIFO( prefix_string + '1', node->rightChild)
end
```

2.3 LookUp

A função *LookUp* recebe como argumentos a "raíz" da árvore de prefixos e um endereço para o qual se quer saber a direção de encaminhamento (o *next-hop*). Caso não esteja definida na árvore a direção de encaminhamento para o endereço requisitado, a função retorna o valor 0 (equivalente a descartar o pacote), caso contrário, retorna o número inteiro correspondente ao *next-hop*. Para endereços inválidos é retornado o valor -1.

A função percorre a árvore sequencialmente, usando o endereço recebido (*bit* a *bit*) para, partindo da "raíz", decidir se "desce" pela esquerda ou pela direita. Em cada iteração, caminha-se para o filho esquerdo caso o próximo *bit* do endereço seja '0', e para o direito caso seja '1'. Existe uma variável na função (que guarda o valor que será retornado) e que é actualizada sempre que o nó corrente possua um *next-hop* definido. Desta forma, caso se termine num nó da árvore que não tenha nenhum *next-hop* definido, a função vai retornar o valor do último *next-hop* pelo qual passou até ali chegar. Assume-se nesta função que caso o endereço possua caracteres diferentes de '0','1' ou tenha mais de 16 *bits*, que este é inválido.

Complexidade: $\mathcal{O}(m)$, onde m = número máximo de bits dos prefixos da tabela de encaminhamento (pior caso).

2.4 InsertPrefix

A função *InsertPrefix* recebe como argumento um prefixo acompanhado de um *next-hop* para ser inserido na tabela de encaminhamento (na árvore virtual utilizada no programa). Caso a árvore esteja vazia, esta é criada com a entrada recebida (mesmo que não exista nenhum encaminhamento de *default*).

O algoritmo processa-se varrendo o prefixo recebido *bit* a *bit* e verificando sempre que o percurso para onde este nos está a levar já existe. Caso não exista um nó na árvore durante este percurso, um novo é criado com um valor de *next-hop* de 0 (não representa nenhuma entrada na tabela de encaminhamento). No final, depois de varridos todos os *bits* do prefixo e caso não se tenha verificado qualquer irregularidade (por exemplo, um *bit* diferente de '0' ou '1') é então inserido o novo *next-hop* associado ao prefixo desejado na árvore.

Complexidade: $\mathcal{O}(n)$, onde n = comprimento do prefixo a inserir na árvore.

2.5 DeletePrefix

Este algoritmo varre cada *bit* do prefixo e anda na árvore de acordo com o que o *bit* lhe indicar da mesma forma como já foi visto em cima. Deste modo, imaginando que se está num nó da árvore e o *bit* agora a processar indica um dos "filhos", ir-se-á primeiramente verificar se o mesmo existe (caso não, retorna-se logo que é um prefixo inválido) e depois disso há algumas condições que se tem de verificar para o correto funcionamento do algoritmo.

Para este algoritmo funcionar corretamente, fez-se uso de um ponteiro e uma variável (inteiro) auxiliar, os quais vão indicar no final a partir de que nó dever-se-á começar a eliminar. Assim sendo, há algumas condicionantes de acordo com as condições do "filho", nomeadamente no que toca aos seus próprios "filhos" ("netos" do nó em questão):

- Este ter outros 2 "filhos" e aí quer-se garantir que este ponteiro auxiliar não se encontra a apontar para qualquer nó, uma vez que este "filho" nunca poderá ser apagado da árvore;
- Este ter apenas um "filho", e aí já há 2 cenários possíveis:
  - Ter o valor de 0 no *next-hop* e neste caso há possibilidade do mesmo vir a ser apagado pois pode vir a ser inútil e nesta situação caso o ponteiro auxiliar não se encontre a apontar para nó nenhum ir-se-á colocar o mesmo a apontar para o pai (duas notas: a primeira é que o ponteiro é posto no pai, pois tem que se garantir que o ponteiro do pai para o filho vai ficar a *null* - daí o inteiro criado que nos indica para que lado tem que se garantir isto e a partir do qual ir-se-á eliminar os nós - e a segunda é que este ponteiro pode já não estar a *null* e aí não irá querer-se mexer pois já se encontra na posição correta);
  - Caso haja um valor no *next-hop* quer-se garantir que o ponteiro auxiliar não se encontra a apontar para nada, pois mais uma vez este filho nunca poderá ser apagado da árvore;

- Este não ter qualquer filho, ou seja, ser uma folha da árvore, repete-se a situação de ter apenas um filho e 0 no *next-hop*.

Posto isto, terminada esta parte do algoritmo basta verificar este ponteiro e a variável auxiliar. No caso em que o ponteiro não esteja a apontar para nó nenhum coloca-se apenas o valor de *next-hop* a 0 no nó correspondente. Caso este esteja a apontar para algum nó a variável auxiliar indica para que lado tem de se começar a eliminar os nós da árvore a partir desse mesmo. Para melhor perceção deste algoritmo, encontra-se em baixo o pseudo-código do mesmo e ainda uma figura em anexo.

Complexidade: $\mathcal{O}(n)$, onde n = comprimento do prefixo a remover na árvore.

Algorithm 2 DeletePrefix(root, prefix[])

```
aux := root
for i := 1 to strlen(prefix) do
    if prefix[i] = '0' then
        if aux → leftChild ≠ null then
            if aux → leftChild has 2 children then
                nodeToStartDeleting := null
            else if aux → leftChild has only 1 child then
                if aux → leftChild → nextHop = 0 then
                    if nodeToStartDeleting = null then nodeToStartDeleting := aux
                else
                    nodeToStartDeleting := null
                end
            else
                if nodeToStartDeleting = null then nodeToStartDeleting := aux
            end
            aux := aux → leftChild
        end
    else
        same behaviour as on the left side
    end
end
if nodeToStartDeleting = null then aux → nextHop := 0
else delete all the nodes from the nodeToStartDeleting
```

2.6 CompressTree

Para a função de *CompressTree* foi implementado o algoritmo de compressão ótima baseado no algoritmo ORTC (*Optimal Routing Table Constructor*) aprensentado no artigo[1].

Todos os passos deste algoritmo foram implementados tendo como base este artigo, contudo o mesmo parte de permissas que podem ser falsas, a tabela de encaminhamento de entrada contém uma *default route* e a tabela contém apenas um *next-hop* por prefixo. A segunda premissa saía fora do contexto do projeto, porém a primeira foi contornada.

No primeiro passo deste algoritmo as "folhas" da árvore foram criados e atribuídos os *next-hops* do parente mais próximo com um *next-hop* diferente de 0, sendo que a presença de uma *default route* garante que todos os "filhos" vão herdar um *next-hop*. Posto isto e como no mundo real pode acontecer existirem tabelas de encaminhamento sem uma *default route*, a técnica utilizada para contornar este problema foi introduzir um *next-hop* fictício (no nosso caso -1). Assim, a tabela de encaminhamento final irá conter entradas com o valor de -1, sendo que estes pacotes devem ser vistos como de um 0 se tratasse e os pacotes descartados. Para compreender com maior rigor o algoritmo implementado, está apresentado em anexo o pseudo-código da função.

Complexidade: $\mathcal{O}(N)$, onde N = número total de *bits* na tabela de encaminhamento (pior caso).

[1] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing Optimal IP Routing Tables. In *Proc. IEEE INFOCOM, 1999.*

Anexo

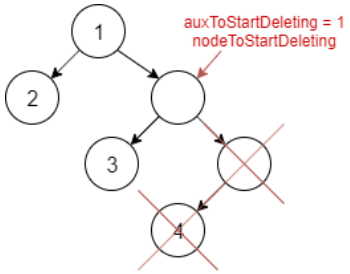

Figura 1: Exemplo do procedimento do algoritmo *DeletePrefix* com os valores das variáveis auxiliares e os nós a eliminar para a eliminção do prefixo correspondete ao *next-hop* 4

Algorithm 3 CompressTree(root)

```
/* ------------- Main Function ------------- */
if root→nextHop = 0 then  root→nextHop := -1
root := PassOneTwo (root)
root→nextHop := root→hopsList→hop
root→leftChild := PassThree (root→leftChild, root→nextHop)
root→rightChild := PassThree (root→rightChild, root→nextHop)

/* ------------- Function PassOneTwo(node) ------------- */
/* step one */
if node is a leaf (0 childs) then
  | insert the parent→nextHop in the child hops list
else if node has only 1 child then
  | create the missing child node
  | insert the parent→nextHop in the child hops list
else
  | if child→nextHop = 0 then
  |   | child→nextHopp := node→nextHop;
  | end
end
node→nextHop := 0
hopsList1 := PassOneTwo(node→rightChild)→hopsList
hopsList2 := PassOneTwo(node→leftChild)→hopsList
/* step two */
node→hopsList := intersect_union(hopsList1, hopsList2)

/* ------------- Function PassThree(node, nextHop) ------------- */
if parent→nextHop ∈ node→hopsList then
  | node→nextHop := 0
else
  | node→nextHop is picked from the node→hopsList
end
if node is a leaf (0 childs) then
  | if node→nextHop = 0 then
  |   | free(node)
  |   | return null
  | end
else
  | node→child := PassThree(node→child, value)
  | node→leftChild := PassThree(node→rightChild, value)
end
```