Instituto Superior Técnico

MEEC

Algoritmia e Desempenho em Redes de Computadores

Tabelas de prefixos e *longest match prefix rule*

Alunos:	Número
Alexandre Filipe	83991
Sofia Salgueiro	84185

Grupo 3

18 de Outubro de 2019

1 Introdução

Na larga rede que é a Internet, no momento que uma mensagem atinge um *router* é necessário que este saiba para que *router* a encaminhar para que esta alcance o seu destino. Isto é conseguido através de tabelas/árvores de endereçamento que, através dos bits do endereço, lidos progressivamente, descodifica qual será o próximo destino da mensagem.

Considerando a velocidade e tráfego da Internet nos tempos modernos é necessária que toda a operação de reencaminhamento seja feita da forma mais rápida e eficiente possível e, tendo isto em mente, foi neste projeto feita a simulação desta operação, tentando analisar e desvendar que procedimentos e algoritmos permitem a tal eficácia desejada.

Todo o código realizado encontra-se em anexo, realizado em C#.

2 Algoritmos

2.1 Criação da árvore

A criação da árvore é realizada no início do programa, lendo um ficheiro .txt, tendo cada linha a informação de um *next-hop*. No momento que um *next-hop* é lido este é adicionado na árvore, descendo até atingir o local que o seu prefixo indica, acrescentando nós vazios caso não haja ainda caminho no sentido em que se desloca.

A operação por completo tem uma complexidade **O(h log(n))**, com h o número de *next-hops* a ler e n o número de nós da árvore, visto cada *next-hop* fazer a sua inserção, uma operação que em média percorre log(n) nós, visto se tratar de uma árvore binária.

2.2 Impressão da árvore

A impressão da árvore, apesar de só necessitar dos h *next-hops*, é preciso percorrer toda a árvore para encontrar estes, resultando numa operação de complexidade **O(n)**. Ponderou-se guardar os valores dos *next-hops* numa lista (visto que podem ser adicionados a qualquer momento é necessária uma estrutura dinâmica), mas como isso iria adicionar a complexidade de inserção e remoção de nós numa lista, além de aumentar a complexidade de outras operações, este método não se achou compensativo e como tal não foi adoptado.

2.3 *Lookup* de um endereço

Para o *Lookup* de um dado endereço desce-se a árvore conforme os *bits* do endereço indica, guardando o valor do último *next-hop* por qual passou. Ao atingir um nó para o qual não tem caminho que continue para onde o endereço aponta, é devolvido o valor do *next-hop* no nó atual ou o valor registado do último *next-hop* que passou pelo caminho. Visto a operação se limitar a atingir um nó numa árvore binária esta tem a complexidade de **O(log(n))**.

Também foi ponderado escrever o valor do último *next-hop* respetivo em todos os nós, deixando de haver nós vazios, mas visto isso trazer complicações à remoção de prefixos e compressão da árvore, esta metodologia não foi aplicada visto a alternativa escolhida só acrescentar a leitura e escrita de um inteiro.

2.4 Inserção de um prefixo

O algoritmo de inserção de um prefixo na ávore é igual ao algoritmo de *Lookup* com a diferença que, ao encontrar um nó que não existe mas que faz parte do prefixo, este é criado e adicionado à àrvore até ser atingido o nó final do prefixo, ao qual é adicionado o valor de *next-hop*. Caso o prefixo já tivesse um valor de *next-hop*, ele é reescrito pelo novo valor. Sendo assim, tal como o algoritmo de *Lookup*, a complexidade é de **O(log(n))**.

2.5 Remoção de um prefixo

Na remoção de um prefixo, à semelhança do *lookup*, é necessário primeiro percorrer a árvore para encontrar o prefixo a remover. Quando se encontra o nó correspondente ao prefixo (caso ele não exista o algoritmo termina simplesmente), o next-hop do nó é apagado (neste caso, corresponde a ter o valor de -1). No entanto, é necessário apagar os nós parentes deste nó que não tenham outros filhos e também não tenham *next-hop.*

Na implementação deste algoritmo utilizou-se recursividade, devido à necessidade de voltar atrás na árvore para apagar enventuais nós antecedentes. Então, para a execução deste algoritmo é necessário, na pior das hipóteses, percorrer a árvore $2 \times \log(n)$ vezes, pelo que a complexidade é de **O(log(n))**. O pseudocódigo deste algoritmo encontra-se em (1).

Algorithm 1 DeleteFunc()

```
nextNode := root
delete := false
for each bit of the prefix from the root to the leaf do
  if bit = '1' then
    nextNode := rightChild
    delete = DeleteFunc()
  else if bit = '0' then
    nextNode := leftChild
    delete = DeleteFunc()
  end if
  if has nextHop or has child then
    delete := false
  end if
end for
if doesn't have a child then
  delete := true
else
  nextHop := -1
end if
return delete
```

2.6 Compressão da árvore

Tal como foi mencionado anteriormente, as tabelas de prefixos encontram-se registadas em *routers*, cuja memória é limitada. Então, pensou-se numa maneira de comprimir as árvores de tal forma a arranjar uma árvore equivalente com o número mínimo de nós. Em [1], é introduzido um algoritmo baseado numa ideia simples e que efetua a compressão de uma maneira eficiente, com complexidade de **O(h log(n))**.

O algoritmo apresentado em [1] é dividido em 3 passos. No entanto, nesta implementação e tal como é também mencionado no artigo, os passos 2 e 3 passaram a ser só um, por forma a diminuir o número de nós visitados. Na figura (1) encontra-se um exemplo da execução deste algoritmo, com a árvore inicial, fig. (1)(a) e a árvore equivalente final, fig.(1)(i).

De uma maneira muito resumida, o algoritmo consiste em, no primeiro passo, obter uma árvore cujos *next-hops* se encontrem apenas nas folhas. Começando pela raiz, o next-hop de cada nó que não seja uma folha é removido e guardado até encontrar um nó seu descendente que apenas tenha 1 filho. Chegando a este nó, é criado o filho que falta, que irá herdar o *next-hop* guardado. Sempre que o algoritmo passa por um nó, não folha, que tem *next-hop*, passa a ser esse o *next-hop* guardado. Este passo corresponde às figuras (1)(b)-(d).

No passo final, começando dos pais das folhas para a raiz, é feita a interseção entre a lista de possíveis *next-hops* dos nós filhos (no caso das folhas, corresponde ao seu *next-hop*). Caso a interseção resulte num ou mais valores, é escolhido um deles para ser o *next-hop* do nó. Posteriormente, os descendentes desse nó que tenham o mesmo *next-hop* são removidos da árvore, para eliminar a redundância. Caso a interseção resulte num conjunto vazio, é registada a união do conjunto como possíveis next-hops(apenas para a execução do algoritmo) e o *next-hop* real continua vazio. Este processo repete-se até chegar à raiz. Este passo corresponde às figuras (1)(e)-(i).

No entanto, apesar de se ter implementado este algoritmo de compressão, as árvores óptimas, ou seja, comprimidas, não são utilizadas. Isto porque, ao comprimir uma árvore, acaba por se perder informação contida na árvore original. Então, remover ou inserir um prefixo na árvore original pode não ser equivalente a remover ou inserir o mesmo prefixo na árvore óptima. Utilizando o exemplo da figura, é fácil verificar que o *next-hop* do prefixo '0' é 2. Então, se este prefixo for removido na árvore original, o *next-hop* de um endereço '011111..' passa a ser 1 mas, se for removido na árvore óptima, o *next-hop* do mesmo endereço passa a ser 5.

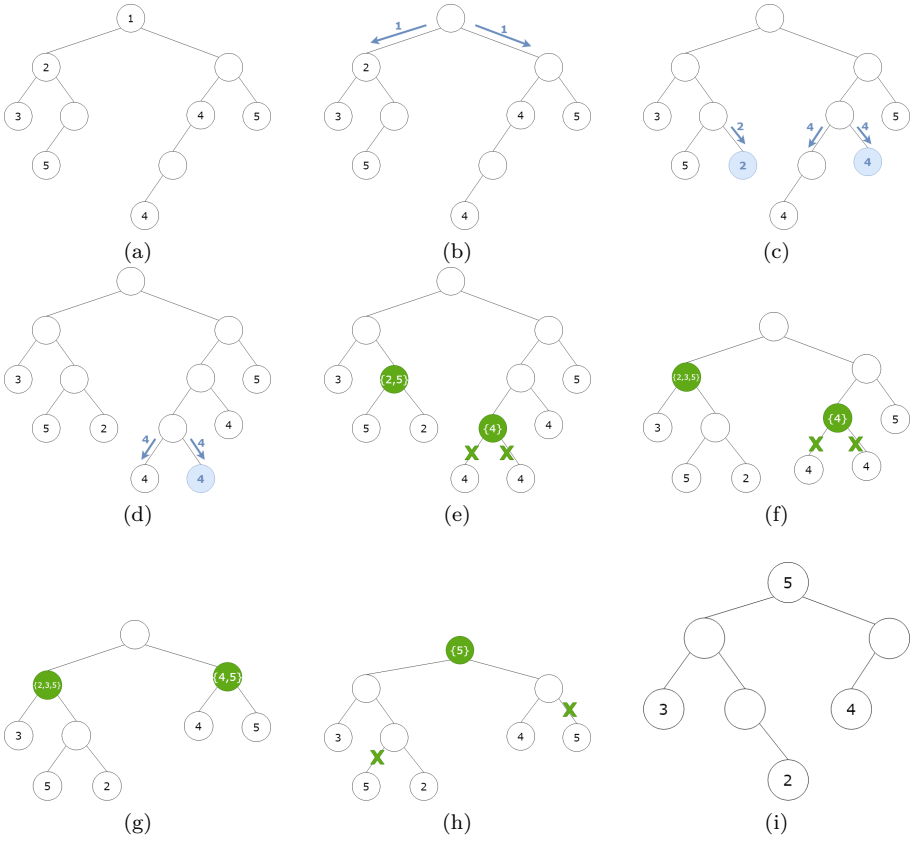

Figura 1: Exemplo de execução da compressão de uma árvore

Referências

[1] Richard P. Draves, Christopher King, Srinivasan Venkatachary e Brian D. Zill. *Constructing Optimal IP Routing Tables*. Proc. IEEE INFOCOM, vol. 1. Mar. 1999, pp. 88–97