INSTITUTO SUPERIOR TÉCNICO

Algoritmia e Desempenho de Redes em Computadores

Projecto 1 - Prefix tables and longest match prefix rule

86984 - Eduardo Rodrigues
87130 - Tomás Malcata

Grupo: 7
18/10/2019

1 Introdução

Neste primeiro projecto, foi estudada a forma como é feito o encaminhamento de endereços de IP num router. Deste modo, foi necessário representar as tabelas de reencaminhamento (presentes em ficheiros de texto) numa árvore de prefixos e posteriormente utilizar a regra do "longest match prefix"na função de procura para ser aferido qual o router vizinho que aproxima mais o pacote de dados ao seu destino final.

Para além do mais, ainda foi pedido para desenvolver alguns algoritmos eficientes com o objectivo de realizar as funções básicas no tratamento destas tabelas, nomeadamente a função de procura do próximo router (LookUp()), acrescento de prefixos (InsertPrefix()), remoção de prefixos (DeletePrefix()), impressão da tabela (PrintTree()) e ainda, com o intuito de optimizar estas funções, a função que comprime uma árvore (CompressTree()). Função esta, com o objectivo de diminuir o número de entradas na tabela de prefixos correspondente de modo a tornar mais rápida a procura, mantendo o mesmo comportamento de reencaminhamento.

Deste modo o objectivo deste trabalho foi o de desenvolver um programa que leia um ficheiro de texto onde esteja presente uma tabela de prefixos e, de seguida, representar essa mesma tabela numa árvore binária de modo a armazenar os dados e posteriormente facilitar as operações sobre estes.

2 Suposições, metodologia e utilização do programa

De modo a desenvolver as funções pretendidas foi necessário tomar por pressuposto diversas idealidades. Entre elas, foram suprimidas certas verificações sobre a má utilização do programa. Assume-se que os prefixos e os ende-reços são sempre inseridos em numeração binária.

Por fim, para a utilização do programa (desenvolvido em linguagem C) é necessário passar como argumentos o nome do ficheiro com a tabela de prefixos e o número máximo de bits dos endereços (Assume-se por simplificação que o número de next hops diferentes é dado por $(Total(nexthops)^2)$ e não por 2^p, onde p é o número de bits do endereço IP).

Sobre a estrutura da tabela do ficheiro, esta deve ter um prefixo por linha, separado de um espaço do next hop correspondente. O prefixo da raiz é representado por um 'e'.

3 Funções principais e análise de custo

Para todas as funções pretendidas é realizada uma procura na árvore em profundidade e é sempre utilizado a recursividade.

3.1 InsertPrefix()

Esta função recebe como argumentos uma string com a informação do prefixo a inserir, um inteiro com o valor do next hop e a raiz de uma árvore de prefixos previamente inicializada.

A função InsertPrefix encontra-se descrita no algoritmo 1.

Facilmente se observa que o crescimento temporal apenas depende do número de bits do prefixo (p), tendo um iteração por cada bit.

Deste modo esta função tem crescimento temporal linear O(p), não dependendo do número de elementos da árvore. Em termos de crescimento do uso de memória, tal é também O(p), sendo que o seu custo diminui (tendendo para O(1)),tanto quanto mais elementos tenha a árvore (devido a já haver nós alocados).

Algorithm 1 InsertPrefix

procedure INSERTPREFIX(tree, prefix, hop, i)
    if prefix(i) = '\0' **then**
        hop(tree) ← hop
        return
    if prefix(i) = '0' **then**
        if zero(tree) = NULL **then**
            zero(tree) ← *new_node*()
        InsertPrefix(zero(tree), prefix, i+1)
        return
    if prefix(i) = '1' **then**
        if one(tree) = NULL **then**
            one(tree) ← *new_node*()
        InsertPrefix(one(tree), prefix, i+1)
        return

3.2 PrefixTree()

Esta função recebe como argumentos uma árvore e o ficheiro de texto com a tabela de prefixos. Deste modo, para cada prefixo lido, este é inserido na árvore através da função InsertPrefix(). Deste modo, em termos de complexidade temporal é de O(n), em que cada iteração a complexidade não é unitária, mas de O(p). Logo, a complexidade temporal total é de O(np).

Por fim, a nivel de memória no pior caso tem-se também O(np), contudo, tendo a árvore cada vez mais preenchida, este valor tende para O(n) (caso o número de nós inicializados com hop a -1 seja nulo).

3.3 PrintTable()

Para a impressão foi criada uma função que recebe como argumentos a árvore e uma string auxiliar. Na árvore é realizada uma busca em profundidade com uma impressão do nó em pré ordem. Para o auxilio da impressão do nó, foi utilizado uma string inicializada a '\0' que a cada iteração altera o valor para o prefixo do nó a imprimir. Deste modo não é necessário alocar uma string por nó com a informação do prefixo, mas uma só para todos os nós. Após a compressão existirão prefixos a '0' ose não tiverem next hop definido. Quanto ao gasto de memória, este é de uma só string auxiliar com dimensão do número máximo de bits. Sobre o gasto temporal, é realizado uma ite-ração por cada nó alocado (mesmo que seja alocado com hop a -1). Sendo então o seu crescimento linear com n, ou seja O(n).

3.4 LookUp()

Esta função recebe como argumento um endereço, uma árvore, um hop (inicialmente negativo) e um inteiro nulo. A árvore é percorrida consoante o valor do bit do endereço, gurdando semore o último next hop encontrado. No final (quando chegar ao fim da árvore) é retornado o valor do next hop para esse endereço. Caso não exista preefixo retorna -1. O pseudo código deste algoritmo encontra-se descrito no algoritmo 2.

A complexidade temporal deste algoritmo é dada por O(p), em que p é o último prefixo alocado encontrado por este endereço na árvore.

Contudo, em termos do número total de elementos e visto que os dados são guardados numa árvore de representação binária é possível considerar que o custo temporal é dado por $O(log_2(n))$, onde n é o número total de elementos.

Em termos de memória é apenas alocado uma string que guarda o valor do endereço, logo o custo é de O(p).

3.5 DeletePrefix()

A função DeletePrefix, tal como o nome indica, recebe um endereço da parte do utilizador

Algorithm 2 LookUp

```
procedure LOOKUP(ip, tree, i, hop)
    if hop(tree) != -1 then
        hop ← hop(tree)
        return
    if (ip(i)='0') && (zero(tree)!=NULL) then
        hop ← LookUp(ip,zero(tree),i+1,hop)
    if (ip(i)='1') && (one(tree)!=NULL) then
        hop ← LookUp(ip,one(tree),i+1,hop)
    return hop
```

e apaga o nó da árvore correspondente a esse mesmo prefixo. Caso este nó não tenha filhos e os seus ascendentes sejam nós sem next-hop predefinido, é necessário ir apagando estes mesmo nós até chegar a um que tenha reencaminhamento para o próximo router. Dito isto, a função recebe uma string com o prefixo a apagar e utilizando a recursividade vai descendo a árvore até chegar ao nó desejado. Posteriormente, caso este nó exista, segue-se a eliminação do mesmo e, devido à recursividade, volta-se à analise do nó pai, apagando os antecessores pelo caminho (caso estes tenham hop a -1 e não tenham filho). Para além do mais, caso o nó a apagar tenha fihos, o hop é apagado com uma simples troca de valor do hop para -1. No fim, caso o prefixo não exista, a função retornará -1.

3.6 CompressTree()

Para a compressão da tabela e para não se perder a informação da tabela original (permitindo assim continuar a apagar prefixos existentes) é feita uma cópia da árvore de prefixos para uma árvore auxiliar. Em todos os passos, a árvore resultante é equivalente à original. De notar que antes de começar caso nã exista um next hop por defeito para a raiz este é colocado a 0 (sem destino). Sendo que na compressão este valor é tratado como um hop e fora como -1. O custo computacional é dado pela soma de todos os nós, sendo portanto $O(n) * 5 * O(w^2)$, ou seja, $O(4 * n * w^2)$, sendo w o número de prefixos diferentes, logo o crescimento é linear.

3.6.1 Passo 1

O primeiro passo efectuado é o da eliminação de prefixos que são prefixos de prefixos com iguais hops. Este tem crescimento O(n), uma vez que passa uma vez por cada nó.

Algorithm 3 DeleteRedundants

```
procedure DELETERED(tree, aux)
    if tree = NULL then return
    prev_hop = hop
    if hop(tree) != hop && hop(tree) != -1 then
        hop ← hop(tree)
    DeleteRed(zero(tree), hop)
    DeleteRed(one(tree), hop)
    if Zero(hop) != prev_hop && prev_hop != -1 then
        if NofSuns(Tree) = 0 then
            free_node(tree)
    else
        hop(tree) ← −1
```

3.6.2 Passo 2

Neste segundo passo é efectuada uma passagem em profundidade pela árvore e uma análise em pré ordem. Este passo tem como propósito colocar todos os hops nas folhas da árvore (sem filhos) e os restantes hops a -1. No final deste passo não existem nós da árvore com apenas um filho. Para tal é necessário percorrer a árvore em profundidade e realizar uma análise em pós ordem. O custo computacional desta função é de O(n), visto que passa uma vez em cada nó.

Algorithm 4 PushToLeaf

```
procedure PUSHTOLEAF(tree, hop)
    if tree = NULL then return
    if Suns(tree)!=0 && hop(tree)!=-1 then
        hop ← hop(tree)
        hop(tree) ← −1
    if Zero(tree) = NUll then
        zero(tree) ← new_node()
        hop(zero(tree)) ← hop
    if One(tree) = NUll then
        one(tree) ← new_node()
        hop(one(tree)) ← hop
    PushToLeaf(zero(tree), hop)
    PushToLeaf(one(tree), hop)
```

3.6.3 Passo 3

O terceiro passo realiza a operação A#B em cada nó. Esta operação é utilizada numa análise em profundidade em pós-ordem da árvore e, em cada nó, guarda num vector os possíveis next-hops para esse mesmo nó. Mais explicitamente, para cada nó, excepto as folhas da árvore, o algoritmo A#B efectua uma de duas operações possíveis. Caso a intersecção dos filhos seja diferente de zero, o valor do next-hop toma o valor desta intersecção, caso contrário, guarda no vector dos possíveis next-hops os valores de ambos os filhos. Computacionalmente esta operção A#B tem um custo quadrático $O(w^2)$, uma vez que a comparação é feita entre todos os elementos do vector A e B. por fim estas comparações são feitas em todos os elementos, tendo um crescimento total de $O(n) * O(w^2)$.

3.6.4 Passo 4

Para a realização deste passo é realizado uma análise em profundidade, em ordem. Deste modo, a raiz tomará o valor do seu primeiro hop possível (guardado no vector). O nó seguintes caso tenha um hop (no vector dos possíveis) igual ao do pai, ficará com este valo. Caso contrário ficará com o valor de um dos hops do vector dos possíveis (por defeito o primeiro). Este crescimento é linear, O(n), uma vez que cada nó é analisado uma só vez.

3.6.5 Passo 5

O passo 5 é igual ao passo 1.

3.7 Conclusão

Com este trabalho foi possível colocar em prática diversos algoritmos relativos à utilização de árvores binária e adaptar um algoritmo de compressão [1]. Conclui-se que numa situação real, esta compressão acaba por ter um custo computacional elevado e um crescimento exponencial da memória utilizada ($O(2^n)$) (em que n são os diferentes next hops possíveis) o que faz com que muitas vezes esta compressão não seja efectuada. Sobre a eficiência da compressão, esta apenas diminuirá o número de prefixos em 70%.

Quanto ao LookUp, este tem um crescimento linear com o numero de bits do endereço, pelo que o custo da operação permanecerá praticamente constante com ou sem a compressão.

Para além do mais, sempre que há uma remoção ou inserção de um novo prefixo ,esta deve ser sempre efectuada na tabela original, pelo que a compressão não é útil nestes casos.

3.8 Referências bibliográficas

[1] R. Draves, C. King, S. Venkatachary, B Zill "Constructing Optimal IP Routing Tables" in Proceedings of INFOCOM '99, New York, March 1999.