Filtrando Intervalos de Endereços IPs

Vicente Vivian *
Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

22 de abril de 2020

Resumo

Neste artigo demonstraremos uma possível solução para reduzir o número de intervalos de endereços IPs em um firewall de rede, mas que também pode ser utilizada para qualquer outro problema cuja fonte de dados sejam intervalos. Começaremos abordando ideias e esboços iniciais sobre qual estrutura de dados pode ser mais simples e eficiente de implementarmos. A implementação possui uma versão inicial que funciona satisfatoriamente e que faz uso do conceito de listas e possui um tempo de processamento dos resultados relativamente pequeno. Uma segunda versão incompleta utilizando Interval Tree foi proposta também.

Além disso, serão discutidos os passos dados até se chegar a esta resolução, assim como de que forma o algoritmo funciona. Junto a isto, uma breve explicação das estruturas de dados e conceitos abordados no texto, e por fim, uma apresentação dos resultados obtidos.

Introdução

A necessidade de resolução do problema pode vir de vários exemplos muito comuns em diversos setores da indústria, neste artigo iremos focar na área de redes de computadores.

Vamos começar pelo o que é um endereço IP. Redes de computadores são formadas e se comunicam através de muitos e muitos endereços IPs. Cada dispositivo conectado em alguma rede possui um IP único por vez, se você deseja se conectar à internet e fazer uso dela, você está usando um IP para tal.

Um firewall é um equipamento com a função de bloquear ou permitir tráfego de uma ou mais redes, junto aos seus endereços IPs. Então, é possível adicionar ao firewall um único endereço IP que você deseja bloquear por qualquer motivo, ou até mesmo um intervalo de endereços. Podemos bloquear qualquer faixa de IPs, como por exemplo do endereço 111 ao endereço 156, assim o acesso a esses IPs ou acessos vindo deles não serão permitidos. Estes intervalos serão representados aqui neste artigo com a notação $[111 - 156]$.

Desta forma, é adicionado ao nosso firewall de exemplo um determinado intervalo de IPs a cada certo tempo a fim de obter uma maior proteção na rede de endereços nocivos. Durante o passar do tempo percebemos que esta lista vai começando a ficar bastante grande, afinal, muito endereços foram adicionados. Quem sabe até mesmo endereços repetidos caso o administrador da rede tenha se perdido em sua imensa lista de intervalos de IPs bloqueados. E aí está o nosso problema, como otimizar e diminuir essa lista? Conseguimos reduzi-la de forma bem significativa ao verificarmos os limites máximos e mínimos dos intervalos e ao aplicarmos algumas espécies de operações de conjuntos como união e intersecção bem como verificar se algum intervalo está parcialmente ou totalmente contido em outro intervalo.

*vicente.vivian@edu.pucrs.br

Vamos abordar algumas ideias conjecturadas que produziram bons resultados e outras que nem tanto para resolver o problema e criar o nosso filtro de IPs. Também iremos demonstrar os tipos de estruturas de dados utilizadas e o algoritmo que realiza a filtragem junto com os resultados de casos de teste buscando testar a eficiência da nossa solução.

Rascunhos de Pensamentos

Diversas ideias de como resolver o problema foram pensadas e teorizadas mas nenhuma com muito sucesso. Então chegou-se a conclusão de parar por um momento com os milhões de pensamentos, raciocinar e ir por partes. Pegar nosso problema relativamente complexo e dividi-lo em problemas menores.

Começando com o mais básico deles, de que forma ler e guardar os diversos intervalos dentro dos arquivos de teste? A linguagem escolhida para construir o filtro de IPs foi Java, pois é a linguagem que o autor está mais familiarizado. Após descrever um algoritmo padrão de leitura de arquivos, era preciso armazenar esses dados em alguma estrutura e depois utilizá-la para aplicarmos o filtro de redução de intervalos. Inicialmente optou-se por utilizar a forma mais natural de solucionar isso, uma lista.

O pontapé inicial foi imaginar guardar cada intervalo em uma lista e posteriormente ir comparando o primeiro intervalo com o segundo, o segundo com o terceiro e assim por diante. Naquele momento, seguir por esta direção resultou na chegada em um beco sem saída do qual honestamente não me recordo qual foi. Então, as próximas estruturas de dados escolhidas para tentarem sua vez foram fila e pilha, estas pareciam serem boas opções para a resolução mas veio o questionamento de que ao não podermos ter acesso a um elemento de qualquer posição nessas estruturas (apenas o último na pilha e o primeiro na fila) a tarefa ficaria bem difícil. A estrutura de árvore parecia ser a mais indicada, porém também a mais complexa, então foi deixada de lado rapidamente e voltamos à estaca zero: a lista.

O que é uma lista, afinal?

Vários termos talvez um pouco abstratos podem ter sido abordados: estruturas de dados pra lá, estruturas de dados pra cá, lista, pilha, fila e árvore. Mas o que é uma estrutura de dados?

Uma estrutura de dados é apenas uma abstração de um certo conceito que neste caso utilizamos para armazenar alguma informação que desejamos, neste momento, queremos armazenar intervalos. Abordando agora primeiramente a estrutura de dados usada nessa implementação do filtro de IPs, a lista.

Uma lista é qualquer sequência de objetos/elementos em qualquer ordem. Todas as estruturas de dados são também uma lista, só que cada uma com sua particularidade. Como a lista usada no meio da programação é uma abstração de um certo conceito, podemos pegar um exemplo utilizando esse conceito visto muitas vezes por nós sobre outra perspectiva.

Vamos supor que você fez uma lista de compras. Você tem um pedaço de papel com diversos elementos anotados em uma dada ordem. Provavelmente essa ordem é a que você vai comprar, do primeiro para o último. No entanto, você pode pular algum elemento (item) e comprar o penúltimo elemento da sua lista caso passe por ele primeiro. Fica a seu critério. Você vai trabalhar com essa lista, seguir a ordem dela, marcar se comprou ou não, anotar o preço etc. A ideia de utilizar listas em programação vem desse tipo de tarefa tão comum nas nossas vidas ou de alguma forma de organizar um afazer do nosso cotidiano. Na próxima página, a Figura 1 demonstra uma representação de um tipo de lista.

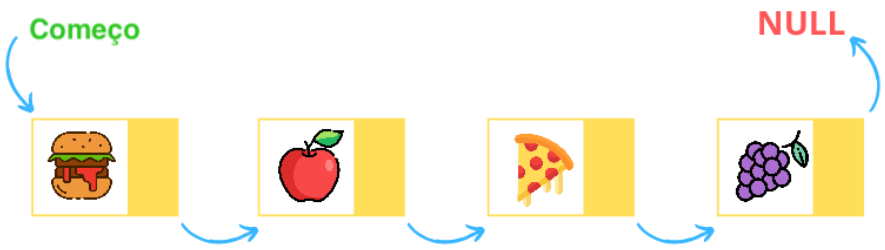

Figura 1: Uma pequena lista de compras.

Alguma coisa definitiva: Filtro de IPs versão 1.0

Apesar de lista ter sido eleita como a estrutura mais simples de obtermos uma solução, chegar em uma resposta com ela não foi tão simples. Recapitulando, temos a leitura do arquivo contendo os intervalos dos IPs feita e os dados foram armazenados em uma lista. Agora vem a parte complicada, como comparar os valores para realizarmos a redução no número de intervalos?

Percebeu-se que a lista deveria estar ordenada de forma crescente para facilitar a nossa situação e executarmos o nosso plano de ataque ao problema em mente. A lista foi ordenada de forma crescente baseando-se no valor mínimo dos intervalos, seguindo esse exemplo: vamos supor que recebemos os valores $[6-7], [3-6], [5-8]$ e montamos nossa lista com esses intervalos. Executando a ordenação na nossa lista, ela ficou com essa cara: $[3-6], [5-8], [6-7]$.

Por sorte, o Java já possui um método (uma forma/função) de ordenação dentro da sua biblioteca de *Collections*, mas é claro que não seria tão simples de utilizá-lo, apesar de já estar pronto para o uso, é necessário dizer ao nosso método como nós desejamos que ele ordene esses pares de intervalos. Para isto foi implementado uma interface chamada *Comparable* nos objetos (elementos) do tipo intervalo, cuja função é informar para o Java e seu método de ordenação o seguinte: ordene esta lista baseando-se em algum critério, neste caso no valor mínimo de cada intervalo. Agora que conseguimos obter uma lista dos intervalos de endereços IPs ordenada, precisamos filtrá-la.

Primeiramente, criamos uma nova lista onde seram armazenados os novos intervalos (chamada de $listaIPsFiltrada$), um objeto do tipo intervalo chamado $intervaloAtual$ e um segundo objeto do tipo Intervalo chamado $proximoIntervalo$. Também iremos nos referenciar à lista original com todos os intervalos lidos como $listaIPsOriginal$.

Agora criamos um laço de repetição (um loop para acessarmos todos os elementos) com seu índice começando em 0 e sendo incrementado por 1 até atingir o tamanho total da $listaIPsOriginal - 1$. Fazendo um breve parênteses, pode parecer estranho irmos até o tamanho da $listaIPsOriginal - 1$, mas isso se deve pela forma como o Java interpreta estruturas de dados, o índice inicial começa em 0 e não em 1, dessa forma uma lista com 10 elementos possui um tamanho 10, mas o índice de seus elementos varia de 0 a 9. A Figura 2 busca esclarecer melhor essa questão.

Figura 2: Indexando nossa lista de compras.

Assim, conseguimos visitar todos os elementos da lista (objetos do tipo intervalo com esse aspecto: $[3-6]$), indo do primeiro elemento da lista até o último. Dentro desse laço, fazemos as devidas comparações a fim de filtrarmos a lista.

Nessa hora está a chave para resolvermos o problema, **a forma de comparar os intervalos**. Pode parecer algo simples de se fazer a princípio, no entanto acabou se tornando um dos maiores desafios. Utilizando a ferramenta de depuração de código construi-se um arquivo de teste que procurasse abordar todos as possíveis variações de acordo com a proposta do problema e a partir disso foi-se executando estes casos aos poucos, construindo as comparações e atribuições conforme cada situação demandava. O algoritmo encontrado para a resolução do nosso problema principal se encontra logo abaixo:

```
1  filtraIPsV1 (Lista< Intervalo>  listaIPsOriginal) retorna tamanho da listaIPsFiltrada
2
3  Cria uma Lista< Intervalo>  listaIPsFiltrada
4  Cria um Intervalo intervaloAtual
5  Cria um Intervalo próximoIntervalo
6  Cria um Intervalo auxiliar ← listaIPsOriginal.elemento(0)
7
8  procedimento filtraIPsV1
9      para índice ← 0 até (tamanho listaIPsOriginal −1) incremente 1 a cada iteração e faça
10
11         intervaloAtual ← auxiliar
12         próximoIntervalo ← listaIPsOriginal.elemento(índice + 1)
13
14         // Caso nenhum limite do próximoIntervalo esteja no intervaloAtual
15         se intervaloAtual.limiteMáximo < próximoIntervalo.limiteMínimo então
16             listaIPsFiltrada.adiciona(auxiliar)
17             auxiliar ← próximoIntervalo
18         fim
19
20         // Caso o próximoIntervalo esteja totalmente contido dentro do intervaloAtual
21         senão se intervaloAtual.limiteMáximo > proximoIntervalo.limiteMáximo
22             auxiliar ← intervaloAtual
23         fim
24
25         // Caso apenas o limite mínimo do próximoIntervalo esteja contido dentro do intervaloAtual
26         senão
27             auxiliar.defineMínimo(intervaloAtual.limiteMínimo)
28             auxiliar.defineMáximo(proximoIntervalo.limiteMáximo)
29         fim
30
31         // Antes de finalizar o laço adiciona o último intervalo àlista
32         se índice = listaIPsOriginal.tamanho −2
33             listaIPsFiltrada.adiciona(auxiliar)
34         fim
35     fim
36     retorna tamanhoListaFiltrada ← listaIPsFiltrada.tamanho
```

Pensamentos soltos sobre uma versão aprimorada

Então, já possuímos uma versão estável que funciona muito bem mas apresenta uma pequena demora ao processar dados muito grandes. Repassando mais uma vez sobre outros tipos de estruturas de dados foi-se observado que um programa utilizando o conceito de árvores aparenta ser mais otimizado e apresenta um melhor desempenho para resolver o problema. Bom, podemos pensar em algo como uma árvore binária de pesquisa (estrutura que facilita a pesquisa e o acesso aos nossos dados) nos sendo mais útil para obtermos os intervalos de menor valor até os de maior. Há uma variação dessa estrutura que pode nos ajudar mais ainda, uma Interval Tree (Árvore de Intervalos).

Interval Tree, o que é você?

Se buscamos responder a essa pergunta, primeiro precisamos esclarecer uma mais básica: do que se trata uma estrutura que armazena dados chamada de árvore?

Uma árvore armazena elementos de maneira hierárquica, essa estrutura recebe esse nome pois ao a olharmos de cabeça para baixo ela nos remete a como você pode imaginar, uma árvore. Essa estrutura de dados é composta por nodos, dentre esses nodos, há o nodo raiz, os nodos internos (galhos) e nodos externos (folhas). A Figura 3 ilustra esse conceito.

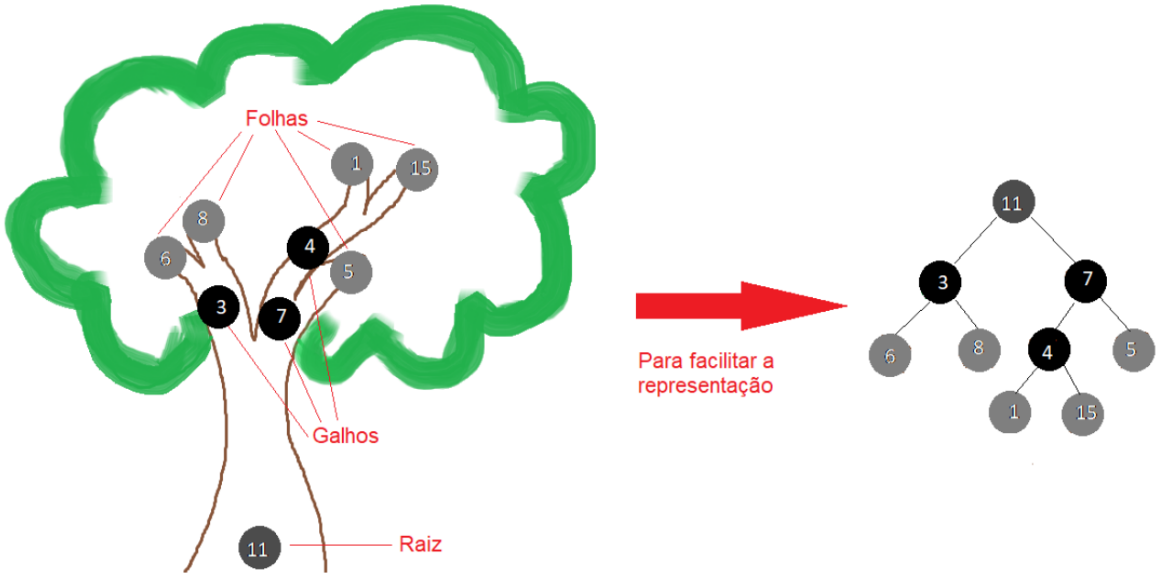

Figura 3: Nossos dados em forma de árvore.

Nosso sistema operacional por exemplo, utiliza a estrutura de árvore para compor nossa hierarquia de diretórios (pastas) dos nossos computadores, como demonstra a Figura 4:

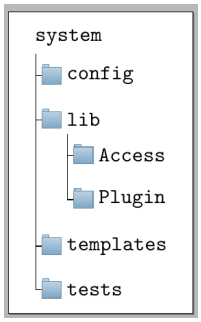

Figura 4: Nossos dados em forma de árvore.

Logo, concluíimos que uma Interval Tree armazena especificamente intervalos em sua estrutura hierárquica de árvore. Por ser uma estrutura específica para o tipo de dados que estamos trabalhando ela fornece operações muito úteis para brincarmos com intervalos. Ela possui diversos métodos interessantes, temos um que busca intervalos que se sobrepoem, outros nos retornam o menor ou maior elemento da árvore. Além disso, a medida que nossos intervalos são armazenados, a Interval Tree já os organiza de forma ordenada, não requerendo que executemos qualquer tipo de ordenação posteriormente.

Algo quase melhor que o definitivo: Filtro de IPs versão 2.0

Bom, a concepção da Interval Tree é genial, ela facilita e aumenta a velocidade do processamento dos dados de maneira impressionante. Como vimos há pouco, ao ler os intervalos, eles são organizados de forma crescente de modo que a árvore fique balanceada– o que significa que nenhum ramo da árvore ficará muito fundo em comparação com os demais. Resultando em um ganho de desempenho bem notável.

Agora precisamos passar esta lista organizada em forma de uma árvore para a nova versão do $filtraIPs$. Se buscarmos seguir a mesma lógica de comparações da primeira versão do programa, acabamos por chegar a um impasse.

Para realizarmos as comparações é preciso obtermos o menor intervalo de todos os elementos junto com seu sucessor. O problema se encontra em acessarmos o sucessor, a estrutura de árvore não permite esse tipo de operação como numa lista. Podemos iterar (percorrer os elementos) sobre a árvore? Sim, podemos, mas apenas de um em um, não é possível termos acesso constante ao elemento atual e ao próximo. Há uma implementação adicional que pode auxiliar no processo chamada *PeekingIterator*, permitindo obter o elemento sucessor, mas ainda assim, uma resolução não foi obtida.

Já se buscarmos ir por outra vertente, fazendo uso dos métodos que a Interval Tree nos fornece também não foi possível chegar a um avanço. É possível filtrar apenas parcialmente os intervalos removendo os elementos que se sobrepoem, mas nada além disso foi concluído.

Resultados

A tabela abaixo apresenta os resultados obtidos ao executarmos o nosso programa. Os valores demonstrados no campo "Tempo de execução"são uma aproximação dos valores, já que há uma pequena variação no tempo obtido a cada execução do programa. Além de representarem a execução do mesmo desde o início da leitura de dados contendo os intervalos até a filtragem dos endereços IPs.

Arquivo	Tamanho da lista original	Tamanho da lista filtrada	Tempo de execução (seg)
caso01.txt	12.000	26	0.286...
caso02.txt	15.000	9	0.183...
caso03.txt	35.000	14	0.335...
caso04.txt	70.000	12	0.340...
caso05.txt	200.000	2	0.910...
caso06.txt	250.000	112	0.911...
caso07.txt	300.000	35	0.909...
caso08.txt	600.000	35	1.246...
caso09.txt	1.200.000	44	2.539...
caso10.txt	2.200.000	64	3.173...
caso11.txt	4.000.000	37	5.233...
caso12.txt	8.000.000	2	10.755...

Conclusões

As abordagens iniciais ao problema, apesar de não oferecerem resultado imediatos, contribuíram de grande forma para o entendimento do mesmo e abriram caminho para a solução definitiva. Esta é bastante simples, mas foi considerada eficiente e deve suprir a necessidade em grande parte dos casos de quem for utilizá-la.

A solução implementada não se preocupou inicialmente em encontrar os menores valores possíveis do tempo de processamento dos dados, apenas buscou resolver o problema em questão, reduzir o número de intervalos de endereços IPs. No entanto, ainda assim os resultados obtidos foram muito satisfatórios e o algoritmo deve ser capaz de ser executado em qualquer máquina comum sem grandes especificações técnicas.

Os resultados de maior importância baseiam-se na redução da quantidade de intervalos, ou seja, o tamanho da lista original em relação ao tamanho da lista filtrada. Após executarmos o nosso método $filtraIPs$ há uma grande diminuição na quantidade de intervalos.

Uma implementação utilizando a estrutura de árvores, mais especificamente sua variação, a Interval Tree aparenta ter uma melhora no desempenho. Porém, durante seu desenvolvimento foram encontrados certas dificuldades com seu progresso ficando estagnado. Permitindo apenas a leitura do arquivo de teste e o armazenamento dos dados de forma ordenada.

Você pode encontrar o repositório com o programa implementado em Java no GitHub, através do link: https://github.com/iVcente/FiltroIPs. Lá é possível baixar, modificar e utilizar o algoritmo aqui descrito.

Referências

[1] Cormen, T. H.; Leiserson, E. C.; Rivest, R. L.; Stein, C.: **"Introduction to Algorithms"**. McGraw Hill Book Co., The MIT Electrical Engineering and Computer Science Series, Cambridge, 2009.

[2] Goodrich, M. T.; Tamassia, R.; Goldwasser, M. H.: **"Data Structures & Algorithms in Java"**. John Wiley & Sons, Inc., Hoboken, New Jersey, 2014.

[3] Vijini Mallawaarachchi: **"8 Common Data Structures every Programmer must know"**. Disponível em: <https://towardsdatascience.com/8-common-data-structures-every-programmer-must-know-171acf6a1a42>. Acesso em: 22 de abril de 2020.

[4] **"C Arrays"**. Disponível em: <https://www.programiz.com/c-programming/c-arrays>. Acesso em: 22 de abril de 2020.

[5] Gonzalo Medina: **"Making a (simple) directory tree"**. Disponível em: <https://tex.stackexchange.com/questions/5073/making-a-simple-directory-tree>. Acesso em: 22 de abril de 2020.

[6] Nélzio Sitoe: **"Base da programação com python : Listas e Tuplas — Capitulo 5"**. Disponível em: <https://medium.com/@nelziositoe/base-da-programação-com-python-listas-e-tuplas-capitulo-5-ec7efee00fc>. Acesso em: 22 de abril de 2020.

[7] **"Estrutura de dados dinâmicas em C - Listas, Filas, Pilhas e Árvores"**. Disponível em: <https://www.cprogressivo.net/2013/10/Estrutura-de-dados-dinamica-em-C-Listas-Filas-Pilhas-Arvores.html>. Acesso em: 22 de abril de 2020.

[8] **"Linked list in C"**. Disponível em: <https://www.programmingsimplified.com/c/data-structures/c-program-implement-linked-list>. Acesso em: 22 de abril de 2020.

[9] Hary Krishnan: **"Peeking Iterator"**. Disponível em: <https://medium.com/@harycane/peeking-iterator-ef69ce9ef788>. Acesso em: 22 de abril de 2020.

[10] Charlotte A Lai: **"Java 8 implementation of a red-black interval-tree, with testing"**. Disponível em: <https://github.com/charcuterie/interval-tree>. Acesso em: 22 de abril de 2020.

[11] Good Ware: **"Free icons designed by Good Ware"**. Disponível em: <https://www.flaticon.com/authors/good-ware>. Acesso em: 22 de abril de 2020.

[12] Freepik: **"Free icons designed by Freepik"**. Disponível em: <https://www.flaticon.com/authors/freepik>. Acesso em: 22 de abril de 2020.

[13] **"Interval Tree"**. Disponível em: <https://www.geeksforgeeks.org/interval-tree/>. Acesso em: 22 de abril de 2020.