Instituto Superior Técnico

MEEC

Algoritmia em Redes e Aplicações

Quality of Service Routing

Alunos:	Número
José Rocha	94304

Grupo 12

24 de outubro de 2022

Conteúdo

1 Introdução

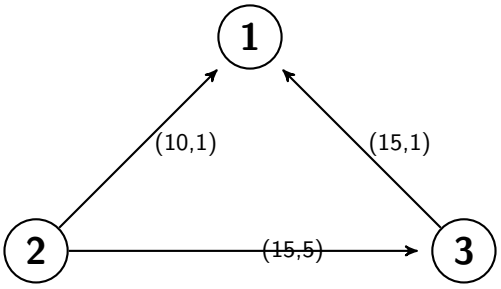
1.1 QoS routing

Na tecnologia de roteamento QoS, cada ligação e cada caminho de uma rede é caracterizado por um par *width-length*. Num caminho, a sua *width* corresponde ao valor da ligação com menor largura; já a *length* de um caminho corresponde à soma dos comprimentos de cada uma das ligações que compõem o caminho.

Entende-se, portanto, que a extensão de um par P *(w; l)* com um outro par Q *(x; y)* resulta no caminho PQ com *(min [w; x]; l + y)* (1).

1.2 Pesquisa de caminhos

Existem dois grandes critérios para escolher os pares *width-length* e os caminhos que dessa escolham resultam: widest-shortest e shortest-widest.



Na rede acima podemos assistir a comportamentos de escolha diferentes dependendo da ordem de eleição dos nós. Por exemplo, supondo que temos como *origem* o nó 2 e como *destino* o nó 1, então teremos dois cenários distintos.

Numa pesquisa widest-shortest, o nó 2 vai escolher a ligação 2-1 com largura 10 e comprimento 1. Isto acontece porque olhando às ligações disponíveis, 2-1 e 2-3, ele escolhe aquela que tem menor comprimento. No caso dos comprimentos serem iguais, então escolheria a que tem maior largura. Como escolheu a ligação 2-1 o nó 2 chegou 1, que é o destino que nos propusémos a encontrar. Podemos então dizer que o caminho mais largo de entre os mais curtos, começando em 2 e acabando em 1, tem como par largura-comprimento *(10, 1)*;

Por outro lado, numa procura shortest-widest o algoritmo vai tentar encontrar os caminhos mais curtos de entre os mais largos, priviligiando aqueles que obtiverem maior largura. Assim sendo, o nó 2 desta feita escolhe a ligação 2-3, com par *(15,5)*, chegando ao nó 3. O nó 3 só tem uma ligação disponível, a ligação 3-1, pelo que

opta por essa ligação. Neste momento o algoritmo extende o caminho que já tinha aprendido, composto apenas pela ligação 2-3. Utilizando a operação de extensão que aprendemos em (1), obtemos o caminho 23- com largura 15 e comprimento 6.

Em ambos os cenários o caminho é encontrado, partindo da origem A e chegando ao destino B, mas como vimos esse caminho encontrado vai depender da ordem de pesquisa que utilizarmos.

1.3 Projeto: desafios e objetivos

Neste projeto desenvolvemos um simulador que propaga mensagens pela rede, acabando por encontrar os caminhos preferidos em relação a ambas as ordens de pesquisa. Adicionalmente, desenvolvemos um algoritmo, assente numa implementação do Algortimo de Dijkstra invertido, onde tentamos encontrar os caminhos pretendidos em cada uma das pesquisas.

Este algoritmo levanta alguns problemas no que diz respeito à procura dos caminhos mais curtos de entre os mais largos, situação que iremos abordar em detalhe mais adiante .

Vamos também fazer algumas comparações entre as métricas que obtivemos via simulação e via algoritmo distribuído, percebendo qual a complexidade e como diferem os resultados que cada um deles obtem.

2 Simulador

Um *QoS vectoring protocol* incia uma computação distinta para cada nó destino. Aqui introduzimos um novo conceito no nosso problema: o conceito de acordar um nó. Cada nó da rede é acordado à vez, iniciando a troca e propagação de mensagens ao longo da rede.

2.1 Matriz de custos

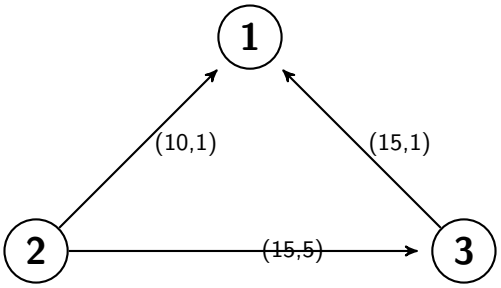
Inicialmente, temos uma matriz NxN onde N é o número de nós da rede. Esta matrix vai permitir-nos guardar o custo da ligação (par *largura-comprimento*) entre cada nó da rede. Por exemplo, na posição (2,3) teremos o custo da ligação entre o nó 2 e o nó 3. Inicialmente, todas as posições têm os valores por defeito (0, 999). Estes são os valores mínimos do nosso programa para a largura e para o comprimentos. São arriscados, podem haver valores superior para o comprimento ou inferiores para a largura, mas estes valores são configuráveis e todo o programa é facilmente adaptável. A ideia destes valores é atribuir ao sistema o pior custo possível, simulando a situação de não haver ligação entre os dois nós. No caso do programa encontrar alguma ligação, ela será sempre melhor que a ligação inicial por defeito e irá optar por essa ligação para extender o caminho procurado.

Caso a execução termine e o custo de uma posição seja o dos valores por defeito, isso significa que não foi encontrada ligação entre os dois nós em questão e a posição da matrix será ignorada.

Acrescentar ainda que um nó, para ele próprio, tem largura máxima de comprimento mínimo, ou seja, tem largura 999 e comprimento 0.

2.2 Execução/troca de mensagens

Relativamente à simulação, cada nó é acordado à vez. Quando o primeiro nó acorda ele envia mensagens aos seus vizinhos internos, os nós que se ligam a ele, comunicando-lhes o custo da sua ligação.



Por exemplo, no caso acima, supondo que o nó 1 é o primeiro a acordar, ele vai enviar mensagens ao nó 2 e ao nó 3, os seus vizinhos internos, comunicando-lhes os custos *(10,1)* e *(15,1)*, respectivamente. Esta execução colocaria *(10,1)* na posição (0,1) da nossa matriz de custos (não é na posição (1,2) porque começamos a contar de 0).

As mensagens não são imediatamente trocadas, são colocadas numa lista ordenada de eventos que aqui vamos apelidar de Calendário. A ordenação assenta no tempo de processamento do evento. Na prática, cada evento é uma das ligações do nó que está a ser acordado para com os seus vizinhos internos. O tempo de processamento de um evento é dado pela soma do instante atual da execução com uma constante fixa e com um valor aleatório resultante da distribuição uniforme entre [0; 1].

Após isto, começa a troca de mensagens propriamente dita através do processamento dos eventos do calendário. Aqui importa referir que o processamento de evento leva ele proprio a que sejam criados novos eventos no calendário, uma vez que à medida que os nós vão sendo acordados vai surgindo a necessidade de comunicar com outros vizinhos internos que, a seu tempo, comunicaraão com outros vizinhos internos propagando a troca de mensagens por toda a rede.

2.3 Processamento eventos

Um evento do calendário contem informação sobre uma ligação. Tendo um evento, podemos saber qual o nó origem, o nó destino, a largura e o comprimento da ligação. É com esta informação que o simulador vai decidir se para o nó origem atual, esta ligação é vantajosa em relação àquela que já encontrou e cuja informação tem guardada na matriz de custos. A decisão é assente na ordem de pesquisa e aqui é fundamental diferenciar a ordem widest-shortest da ordem shortest-widest.

No capítulo anterior já vimos teoricamente como é feita a decisão (1.2), como tal aqui vamos deixar apenas uma anotação de como o simulador opera perante estas circunstâncias.

Supondo que estamos a contrar os caminhos mais curtos a partir do nó X. Já foram executadas uma série de eventos e e o evento atual, o evento E, temos nó origem A, nó destino B e o custo é dado pelo par *(w1,l1)*. Como o nó source é o nó A, este já foi encontrado a partir de X. Vamos supôr que o caminho atual entre XA é dado por *(w2,l2)*, e vamos supôr ainda que ainda não existe caminho entre XB, pelo que a posição em questão da nossa matrix tem os valores por defeito (0, 999), que representamos como o para *(w3, l3)*;

2.3.1 Widest-Shortest

dv = l3, du = l2, luv = l1;
dw = w3, w = l2, wuv = l1;

if ($dv > du + luv$ or ($dv = du + luv$ and $dw < \min(w, wuv)$))
  matrix[X][B].width = min (w, wuv);
  matrix[X][B].length = du + luv;
  parent[X][B] = A;

A aresta AB extende o caminho XAB caso este seja o melhor caminho XB. Se isso acontecer, a matriz de custos é atualizada e o nó A para a ser predecessor do nó B.

2.3.2 Shortest-Widest

dv = l3, du = l2, luv = l1;
dw = w3, w = l2, wuv = l1;

if ($\min(w, wuv) > dw$ or $\min(w, wuv) = dw$ and $dv > du + luv$)
  matrix[X][B].width = min (w, wuv);
  matrix[X][B].length = du + luv;
  parent[X][B] = A;

A aresta AB extende o caminho XAB caso este seja o melhor caminho XB. Se isso acontecer, a matriz de custos é atualizada e o nó A para a ser predecessor do nó B.

2.4 Terminação

A execução do simulador termina quando já não houverem mensagens a processar. Nesse momento teremos na matriz de custos o custo do caminho entre cada nó da rede e na matriz de predecessores os nós que são pais de cada nós de cada caminho. Nessa matrix, alinha 0 representa os pais dos nós (coluna) para o destino 0. Inicialmente esta matrix tinha valor por defeito -1 em cada posição, portanto, no final da execução, onde encontrarmos esse valor significa que não encontrámos nenhum predecessor.

3 Algoritmo

A motivação para desenvolver o algoritmo prende-se com a complexidade do simulador que é exponencial ao número de nós da rede. Isto torna-se muito pouco prático para redes complexas e com muitos nós.

O algoritmo que implementámos é uma adaptação ao nosso problema do algoritmo de Dijkstra invertido. O algoritmo será percorrido N vezes, onde N é o número de nós da rede. Em cada execução será passado como argumento o nó destino e o algoritmo encontrará os caminhos óptimos de cada destino para todas as origens. No final das N iterações, todos os nós terão os seus caminhos encontrados.

Como não implementámos a min heap e trabalhamos com um vector ordenado, essa ordenação da Queue aumenta a complexidade do algoritmo para $O(n^2)$.

3.1 Execução

Inicialmente, a nossa matrix de custos e a nossa matrix de predecessores estarão com os valores por defeito. O nó destino terá como valor para si próprio o par *(999,0)*. Isto é particularmente relevante uma vez que como todos os outros nós têm custos piores, este será o primeiro a ser extraído da queue Q, que no iníco terá todos os nós da rede.

Durante a execução, vamos sempre retirar de Q o nó cujo custo é mais baixo. Para esse nó, exploramos todos os vizinhos internos, tal como fazíamos no simulador, e para cada um deles extendemos o custo da ligação atual com o caminho que já conhecmos desde o nó destino até ao nó que retirámos de Q.

Também aqui, tal como no simulador, vamos utilizar os comparadores indicados nos pontos 2.3.1 e 2.3.2 e atualizamos a matriz de custos e a matriz de predecessores em conformidade.

Quando todas as arestas do nó retirado foram analisadas e relaxadas, o nó é retirado da Queue e o algoritmo prossegue.

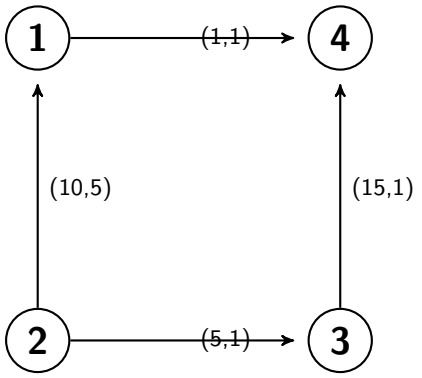
3.2 Terminação

O algoritmo termina quando todos os nós foram retirados da Queue. O estado final da matriz de custos representa os caminhos obtidos e a matriz de predecessores, para cada linha L, guarda os pais dos nós coluna para o destino L.

3.3 Widest-Shortest paths

Como analisámos anteriormente, devido à quebra da isotonicidade, e apesar de encontrarem os mesmo caminhos, o algoritmo de Dijkstra e o Simulador não encontram os caminhos corretos.

Criámos um algoritmo, que não é mais que uma adaptação do algoritmo de Dijkstra utilizado, onde a grande questão residia em perceber o porquê de não encontramos os caminhos ideais.



Por exemplo, nesta rede, quando o nó 2 está a tentar aprender o caminho mais curto de entre os mais largos para o nó 4. Ele vai aprender o caminho 214 quando na verdade deveria aprender o caminho 231.

Isto acontece porque vai de uma forma *ganaciosa* escolher o nó 1 ao invés do 3, que tendo pior largura, é ignorado.

Para contornar esta situação vamos adicionar um conceito ao nosso programa: o par candidato. Neste caso 23 seria um par candidato porque, tendo pior largura que 21, tem menor comprimento. Assim, no futuro, quando o caminho 21 estiver a estudar a aresta 14, perceberá que isto o prejudica, na medida em que o par candidato ignorado anteriormente, 23, lhe oferecia uma melhor opção.

3.3.1 Par candidato

Estamos na presença de um par candidato quando dois caminhos são incomparáveis. Dois caminhos só são comparáveis caso um deles tenha maior largura e menor distância do que o outro. Se estas duas condições não se verificarem, estamos na presença de um par candidato

dv = l3, du = l2, luv = l1;
dw = w3, w = l2, wuv = l1;

if ($dv > du + luv$ or ($dv = du + luv$ and $dw > \min(w, wuv)$))
    matrix[X][B].width = min (w, wuv);
    matrix[X][B].length = du + luv;
    parent[X][B] = A;

Na prática, a execução do algoritmo segue as linhas orientadoras do Algoritmo de Dijkstra, mas quando verifica que existe um par candidato, adiciona-o a uma nova estrutura de dados onde guardamos os pares candidatos para cada nó (no âmbito da pesquisa para um destino). Quando cada nó só tem um par candidato (correspondente ao caminho atual), ele faz o relaxamento das arestas tal como acontecia no algoritmo de Dijkstra.

No entanto, quando existem vários pares candidatos, temos de analisar cada um deles para percebermos se efetivamente melhoram o caminho atual e se devemos alterar as nossas escolhas.

Esta implementação não ficou totalmente finalizada, sobretudo no que diz respeito aos predecessores, de qualquer forma na rede inicial do enunciado conseguimos obter os caminhos mais curtos de entre os mais largos e com a proposição aqui enunciada parece-nos que seria viável considerar esta condição para uma pesquisa efetivamente correta.

4 Modo Interativo

Tanto o simulador como o Algoritmo de Dijkstra têm implementado um modo interativo onde é passado um nó origem e um nó destino para os quais é encontrado o caminho mais curto de entre os mais largos e o caminho mais largo de entre os mais curtos.

5 Conclusão

5.1 Análise

Tanto o algoritmo de Dijkstra invertido como o simulador obtêm os mesmo caminhos com a particularidade de nem um nem o outro obterem os caminhos corretos quando procuramos os caminhos mais curtos de entre os mais largos (shortest-widest). Isto prende-se com a quebra de isotonicidade que se verifica quando analisamos caminhos óptimos, veremos essa questão mais adiante.

5.2 Resultados

5.2.1 Custo dos caminhos

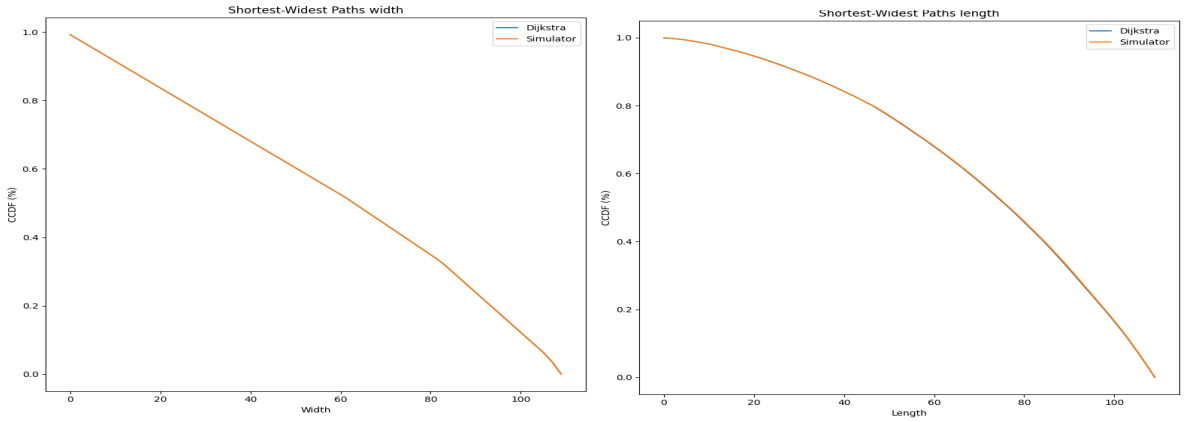


Figura 1: Rede Abilene

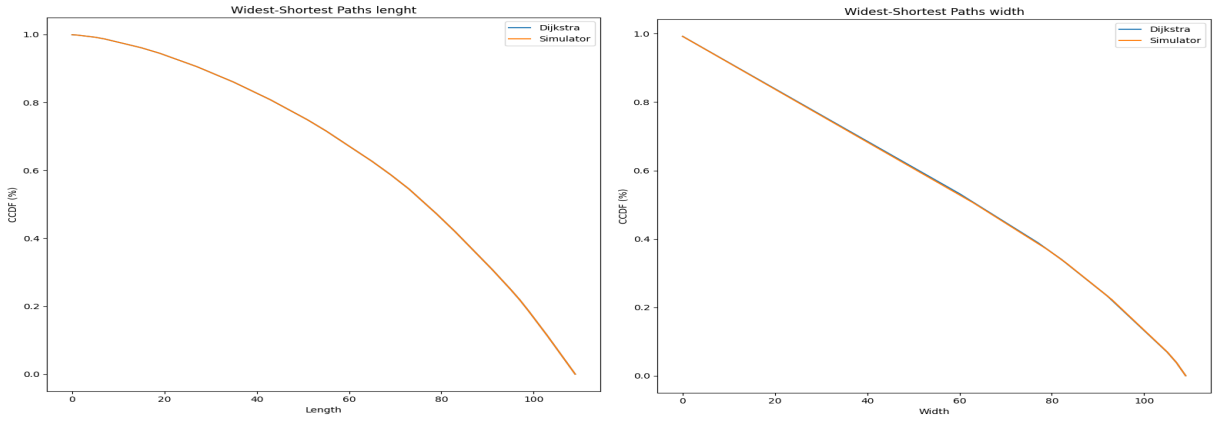


Figura 2: Rede Abilene

5.2.2 Tempos de execução

Olhando à função CCDF dos tempos de execução, percebemos bem que o tempo de convergência dos caminhos está inevitavelmente ligado à dimensão da rede.

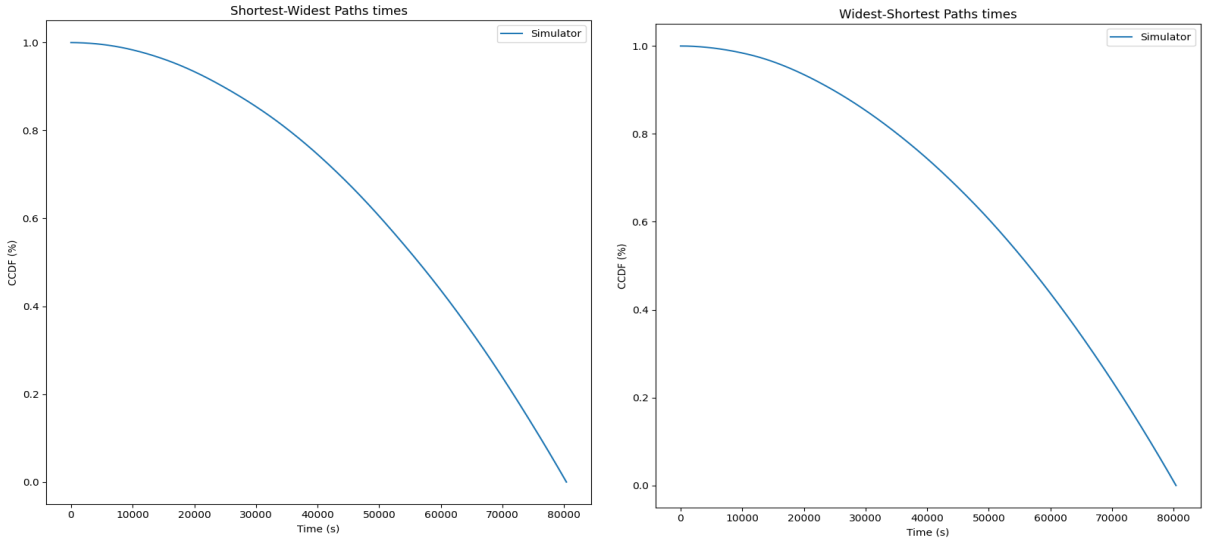


Figura 3: Rede AS1239

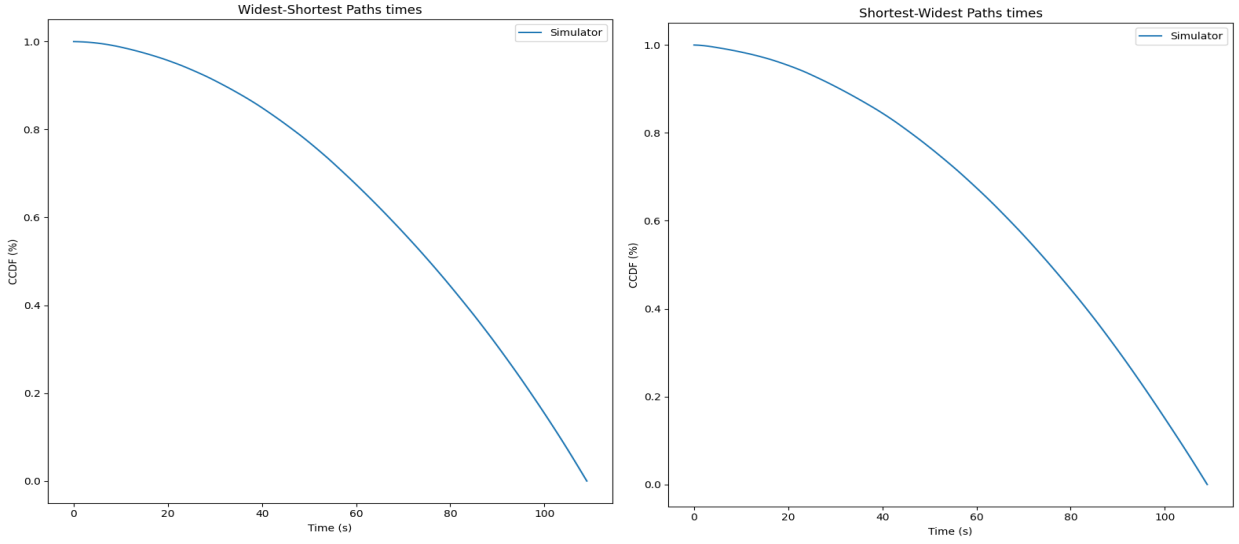


Figura 4: Rede Abilene

5.3 Comentários

Tendo em conta os resultados obtidos, penso que confirmámos duas premissas fundamentais que esperávamos: 1) a complexidade da rede aumenta os tempos de convergência para encontrar os caminhos; 2) os caminhos encontrados por Dijsktra e pelo protocolo de simulação são iguais.

Uma das situações que se verificou tanto no simulador como no algoritmo de Dijkstra prende-se com os tempos de convergência serem equivalentes entre os caminhos mais curtos de entre os mais largos e os caminhos mais largos de entre os mais curtos. Algo que também faz todo o sentido, uma vez que a diferença reside apenas no processo comparativo. No nosso algoritmo corretivo já não será assim, uma vez que com o conceito dos pares candidatos, o número de possíveis comparações será muito maior.

Em relação ao algoritmo, e sobretudo no que diz respeito à implementação, ficou claramente em falta a adequação da estrtura de dados que guarda os predecessores de um nó e que no final nos permite imprimir todos os nós que compõem um caminho. Na implementação do algoritmo faltou mais algum tempo para fazer melhores testes pois nas redes de maior dimensão não foi possível analisar todas as situações para perceber se os resultados estavam completamente corretos.