Instituto de Matemática e Estatística
Universidade de São Paulo

Grafos Evolutivos e Redes Tolerantes a Atrasos

Alunos: César Gamboa Machado
Paulo Henrique Floriano
Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

30 de novembro de 2009

Resumo

Atualmente, existem várias redes móveis com características dinâmicas em funcionamento. Neste cenário, há grande intermitência nas conexões entre os nós da rede, portanto os protocolos de roteamento comuns não apresentam bons resultados.

Para resolver o problema do roteamento de pacotes em redes deste tipo, diversos algoritmos foram propostos. A maioria destes assume que a topologia da rede não é conhecida e, portanto, baseia-se em métodos probabilísticos para decidir se, em certo momento, transmite ou não uma certa mensagem para um certo nó.

Estes algoritmos apresentam bons resultados em cenários práticos, porém, as rotas encontradas nem sempre são as melhores possíveis. Além disso, alguns algoritmos podem gerar muitas cópias da mesma mensagem, possivelmente sobrecarregando a rede.

Para tentar encontrar as rotas ótimas em um cenário teórico, assumimos que a topologia da rede é conhecida e a modelamos com um Grafo Evolutivo, um grafo cujas arestas existem em certos intervalos de tempo. Sobre este modelo, encontramos jornadas (o equivalente a caminhos em um grafo evolutivo) que otimizam certos parâmetros, como o tempo de chegada, o número de saltos e o tempo de trânsito.

Utilizando tais jornadas, podemos obter um roteamento que não replica nenhuma mensagem, diminuindo consideravelmente a carga da rede, levando grande vantagem sobre os algoritmos tradicionais nos cenários estáticos.

Observação

A pesquisa sobre DTNs foi feita em um grupo de cinco alunos. Este grupo consiste dos alunos Adriano Tabarelli, Caio Cestari Silva, Cássia Garcia Ferreira, César Gamboa Machado e Paulo Henrique Floriano. Apesar de termos dividido os assuntos pesquisados para o desenvolvimento das monografias em grupos menores, todos têm uma base em comum.

Sumário

Parte I

Parte Técnica

1 Introdução

Redes Tolerantes a Atrasos (em inglês, *Delay Tolerant Networks*, ou DTNs)[OMR+07] são redes que sofrem grandes mudanças de topologia e tem baixa conectividade. Em redes assim, desconexões e atrasos na entrega de mensagens são frequentes, fazendo com que protocolos de rede comuns como o TCP/IP se tornem ineficazes.

Exemplos de DTNs incluem redes móveis ad-hoc, redes de satélites e a planejada Internet Interplanetária.

Um problema comum em redes assim é como entregar mensagens entre dois pontos de forma eficiente.

Existem diversos algoritmos de roteamento para DTNs em que não há informações sobre conexões futuras. Exemplos de algoritmos assim são o roteamento epidêmico [VB00], o spray-and-wait [SPR05], o PRoPHET [LDS04] e o MaxProp [BGJL06]. Tais algoritmos geralmente criam várias cópias da mesma mensagem na rede, de forma a aumentar a probabilidade de entrega, porém, existe uma possibilidade de sobrecarga da rede devido ao grande número de mensagens circulando.

Entretanto, se temos informações sobre todas as conexões que irão ocorrer na rede (como no caso da Internet Interplanetária, por exemplo), é possível encontrar rotas ótimas para as mensagens, diminuindo o número de pacotes perdidos e de cópias de uma mesma mensagem transitando na rede. Para isso, precisamos de uma estrutura que guarde tais informações.

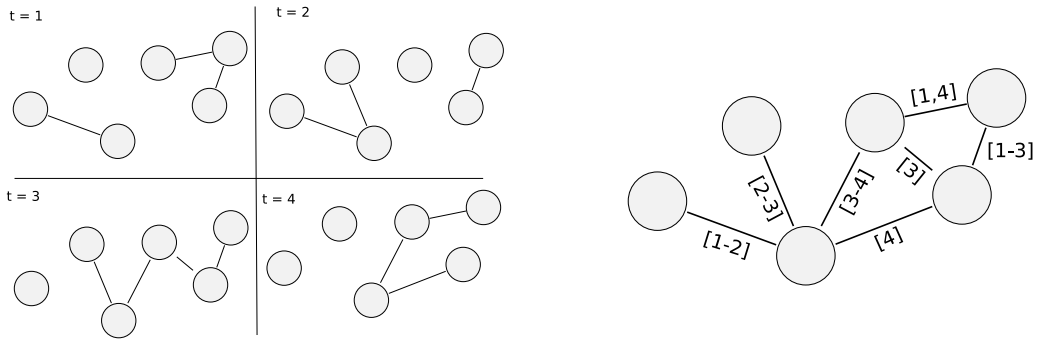
Grafos Evolutivos são grafos em que cada arco possui uma lista de intervalos de tempo nos quais este está ativo. Tais estruturas são muito úteis para a representação de DTNs previsíveis, nas quais se conhece o tempo de início e o tempo de término de cada conexão entre nós.

A figura 1(b) mostra um exemplo de Grafo Evolutivo.

Em um Grafo Evolutivo a noção de caminho deve ser alterada para representar o tempo de utilização de cada aresta (obviamente, uma aresta não pode ser percorrida em um tempo anterior ao tempo de transição da última aresta). Um caminho em que cada aresta tem um instante de tempo no qual deve ser percorrida é chamado de Jornada.

O conceito de custo de caminhos pode ser estendido para jornadas sob a forma de três parâmetros diferentes:

- O instante no qual a mensagem é entregue. Uma jornada que otimiza este parâmetro



(a) Conexões em quatro instantes de tempo.

(b) Grafo evolutivo correspondente a 1(a)

Figura 1: Conexões e Grafo Evolutivo correspondente

é chamada *Foremost Journey*.

- O número de arestas utilizadas na jornada. Uma jornada que utiliza o número mínimo de arestas é chamada *Shortest Journey*.

- O tempo de trânsito da jornada (a diferença entre o tempo de chegada e o tempo de início). Uma jornada que otimiza o tempo de trânsito é chamada *Fastest Journey*.

Algoritmos para calcular essas três jornadas foram propostos em [XFJ02]. No entanto, o algoritmo *Fastest Journey* nunca havia sido implementado ou usado em contextos reais.

O objetivo deste trabalho é o estudo dos algoritmos conhecidos para encontrar jornadas em Grafos Evolutivos, e a comparação destes com os algoritmos para entrega de mensagens em DTNs onde a rede não é toda conhecida.

Para isso, implementamos os algoritmos de jornadas em Grafos Evolutivos (*Foremost Journey*, *Shortest Journey* e *Fastest Journey*), comparando sua eficiência com a dos algoritmos de roteamento comuns com o auxílio do simulador de redes ONE (Opportunistic Network Environment simulator) [KOK09].

Nas próximas seções, descreveremos, em detalhes, uma Rede Tolerante a Atrasos e Desconexões, apresentaremos as definições formais de Grafo Evolutivo e Jornada, apresentaremos os algoritmos para cálculo de jornadas ótimas, bem como justificativas de sua correção. Finalmente, descreveremos os cenários de teste utilizados no simulador e apresentaremos os resultados encontrados.

2 Redes Tolerantes a Atrasos

Em redes fixas, a arquitetura da Internet é uma solução tecnológica de comprovado sucesso, pois seus protocolos são robustos, eficientes e flexíveis. Porém, em cenários nos quais os nós da rede se movem, provocando atrasos na entrega de pacotes ou intermitência nas conexões, tal arquitetura mostra-se ineficaz, pois depende de uma conexão fim-a-fim entre a origem e o destino da mensagem.

Redes com tais características são chamadas Redes Tolerantes a Atrasos e Desconexões (*Delay and Disruption Tolerant Networks*, ou DTNs). As duas principais características de DTNs são:

- Atrasos variáveis: Uma mensagem transmitida de um nó para um vizinho pode não chegar instantaneamente, devido à distância entre os nós, à largura da banda ou ao tamanho da mensagem.

- Conexões intermitentes: Devido à movimentação dos nós, uma conexão que está aberta num dado instante pode não estar mais no instante seguinte, dependendo do raio de transmissão, da distância e da velocidade dos nós.

Para contornar estes problemas, os nós de uma DTN não podem simplesmente retransmitir os pacotes assim que os recebem, pois a conexão que leva a mensagem ao destino pode demorar a se abrir. Além disso, não se pode esperar que exista uma conexão perene entre todos os nós no caminho da origem da mensagem até seu destino para que se inicie uma transmissão. Ao invés disso, cada nó possui um tipo de armazenamento persistente de dados para guardar as mensagens que devem ser transmitidas no futuro. Deste modo, os pacotes não são enviados diretamente para o receptor, e sim, armazenados e reenviados para cada nó do caminho, como mostra a figura 2.

Alguns cenários de redes que apresentam estas características são: redes de sensores sem fio, redes móveis ad-hoc (Mobile Ad-hoc Networks - MANETs) e etc. Em alguns locais onde é muito custoso para se levar internet banda larga comum, como na região da Lapônia, a tecnologia DTN é utilizada. Os dados enviados pelas pessoas ficam armazenados em um dispositivo e, em certas horas do dia, um helicóptero ou outro veículo passa, recolhe estes dados automaticamente e os leva para um local no qual existe conexão com a internet. Posteriormente, o veículo recolhe as respostas das requisições e as leva de volta para o local remoto.

Neste tipo de rede, o problema em que estamos interessados é o de transmitir uma mensagem de um ponto a outro de forma eficiente. Falaremos agora um pouco sobre os algoritmos

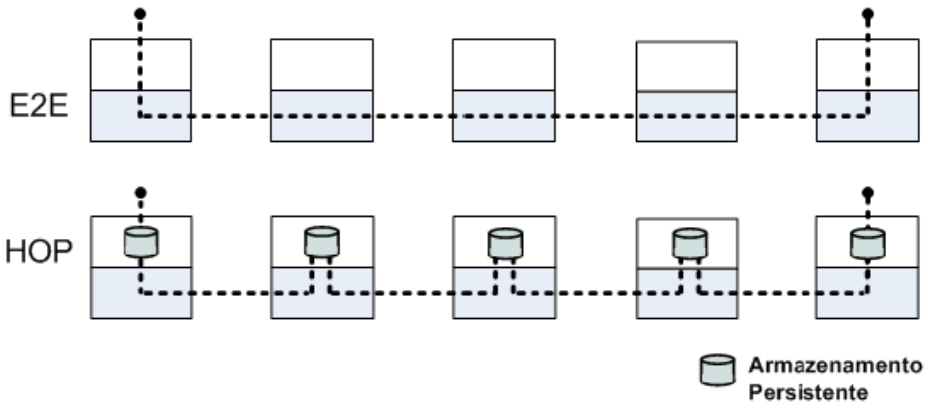


Figura 2: Transmissão fim-a-fim (E2E) e transmissão em DTNs (HOP). Figura retirada de [OMR+07].

de roteamento em DTNs que não consideram nenhum conhecimento sobre a topologia da rede.

2.1 Algoritmos de Roteamento

Para resolver o problema da entrega de mensagens sem nenhum tipo de hipótese sobre a rede, mas, ainda assim, garantindo uma certa probabilidade de entrega, não há outra opção senão gerar mais de uma cópia da mesma mensagem e repassar para nós diferentes.

O algoritmo mais simples para este problema, chamado de Epidêmico, consiste em, a cada contato de um nó que carrega a mensagem com um outro que não a carrega, uma nova cópia desta é criada e transmitida. Desta forma, todos os nós alcançáveis a partir daquele no qual se originou a mensagem receberão uma cópia desta, se a capacidade de armazenamento de cada nó for infinita. Mas, esta suposição está longe de ser aceitável em cenários reais, onde podem existir milhares de mensagens diferentes circulando pela rede. Se diversas cópias forem geradas para cada uma, certamente haverá grande número de estouros de *buffer* e muitas das cópias podem ser perdidas. Além disso, a quantidade enorme de mensagens que seriam geradas certamente acarretaria na sobrecarga do sistema.

Para diminuir o número de retransmissões da mesma mensagem, pode-se considerar o histórico de encontros entre os nós e retransmitir uma mensagem apenas para um nó que tenha uma probabilidade alta de entregá-la ao seu destino. Baseado nesta ideia funciona o algoritmo chamado PRoPHET (*Probabilistic Routing Protocol using History of Encounters and Transitivity*). Cada nó a mantém uma lista que informa, para cada outro nó b da rede, a probabilidade de uma mensagem armazenada em a ser entregue para b. Os valores desta lista são atualizados quando dois nós se encontram (a probabilidade de um entregar para o outro aumenta) ou quando muito tempo se passa sem que dois nós se encontrem (a

probabilidade de um entregar para o outro diminui). Além disso, os valores são atualizados por transitividade, ou seja, se a encontra b frequentemente e b encontra c frequentemente, então, a probabilidade de entrega de a para c também deve aumentar. A estratégia de encaminhamento das mensagens é simples: quando dois nós se encontram, a mensagem em questão fica com o nó que possui maior probabilidade de entregá-la para seu destino. Desta forma, geram-se poucas cópias de cada mensagem e, em cenários onde a movimentação dos nós se repete frequentemente, pode-se garantir uma alta probabilidade de entrega.

Uma variante deste algoritmo é o chamado MaxProp. Neste protocolo, cada nó mantém a lista com as probabilidades de entrega para os outros nós da rede, mas a estratégia de roteamento é diferente. A cada mensagem guardada por cada nó, é atribuído um custo que é uma estimativa de sua probabilidade de entrega. Aquelas que foram criadas mais recentemente tem prioridade maior. Neste algoritmo, usam-se também os chamados *ACKs* para notificar os nós da entrega de pacotes e permitir que as cópias desnecessárias destes sejam apagadas para liberar espaço para mensagens novas. Desta forma, quando há uma conexão, os nós trocam seus valores de probabilidades e calculam a probabilidade de entrega de cada pacote. Assim, pacotes com maior probabilidade de entrega (e, consequentemente, maior prioridade) são transmitidos primeiro e os pacotes com menor probabilidade de entrega são apagados primeiro caso o *buffer* fique cheio.

Diversos outros protocolos de roteamento foram criados para DTNs, mas todos utilizam dados probabilísticos para determinar os encaminhamentos. Apesar disso, os resultados obtidos são satisfatórios, pois vários deles conseguem alcançar altas probabilidades de entrega e relativamente baixa latência.

3 Grafos Evolutivos

Em alguns casos, é possível saber de antemão todas as conexões que ocorrerão ao longo do tempo na rede. Em tais situações, podemos modelar a DTN com um grafo que varia ao longo do tempo.

3.1 Modelo Teórico

Definição 1 (Grafos Evolutivos)**.** *Sejam $G = (V_G, E_G)$ um grafo e $S_G = G_0, G_1, \ldots, G_\tau$ ($\tau \in \mathbb{N}$) um conjunto ordenado de subgrafos de G tal que $\bigcup_{i=1}^{\tau} G_i = G$. O sistema $\mathcal{G} = (G, S_G)$ é chamado de Grafo Evolutivo.*

Seja $V_\mathcal{G} = \bigcup V_i$ e $E_\mathcal{G} = \bigcup E_i$. Definimos $N = |V_\mathcal{G}|$ e $M = |E_\mathcal{G}|$. Dois vértices são adjacentes em $\mathcal{G}$ se, e somente se, são adjacentes em algum G_i. O tempo de duração da transmissão de uma mensagem por uma aresta e do grafo é dado por $\zeta(e)$.

Adicionalmente, para cada aresta e de $E_\mathcal{G}$, podemos definir seu horário de atividade $P(e)$, que corresponde a uma lista de intervalos de tempo em que se pode atravessá-la. Definimos $\delta_E = max\{|P(e)|, e \in E_\mathcal{G}\}$, o maior dos conjuntos de intervalos no grafo e $\mathcal{M} = \sum_{e \in E_\mathcal{G}} |P(e)|$, o número de intervalos do grafo. Para simplificar os cálculos, assumimos que todos os intervalos em $P(e)$ são maiores que $\zeta(e)$.

3.2 Jornadas

Neste modelo, para encontrarmos uma rota para uma mensagem, não basta calcularmos um caminho em G, já que, como podemos observar na figura 3, este poderia, eventualmente, utilizar uma aresta que existiu apenas no passado. Precisamos estender o conceito de caminho para considerar o tempo de percurso de cada aresta.

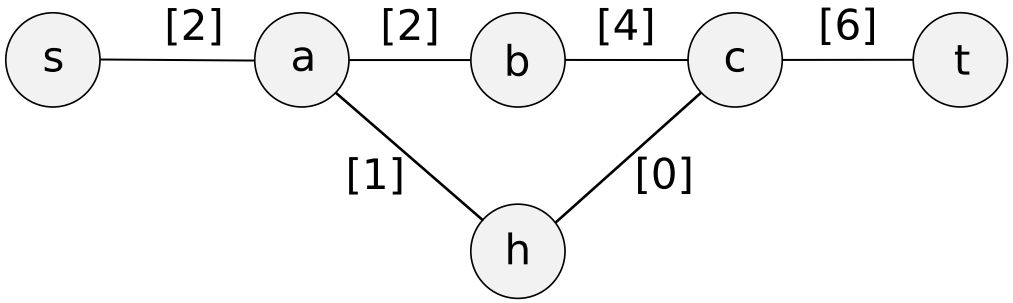


Figura 3: $s \xrightarrow{2} a \xrightarrow{2} b \xrightarrow{4} c \xrightarrow{6} t$ é uma jornada válida, mas $s \xrightarrow{2} a \xrightarrow{1} h \xrightarrow{0} c \xrightarrow{6} t$ não

Definição 2 (Jornada)**.** *Seja $R = e_1, e_2, \ldots, e_k$ ($e_i \in E_G$) um caminho em G, e $R_\sigma = \sigma_1, \sigma_2, \ldots, \sigma_k$ ($\sigma_i \in [1, \tau]$, $\sigma_i \leq \sigma_j \ \forall i < j$) um agendamento indicando quando cada arco de R deve ser atravessado. Então dizemos que $\mathcal{J} = (R, R_\sigma)$ é uma jornada em $\mathcal{G}$.*

Definimos, também, para uma jornada $\mathcal{J}$, $arrival(\mathcal{J})$ como o instante de chegada ao destino ($\sigma_{|R|} + \zeta(e_{|R|})$), $departure(\mathcal{J})$ como o instante de utilização da primeira aresta (σ_1) e $transit(\mathcal{J})$ como o tempo de trânsito da jornada ($arrival(\mathcal{J}) - departure(\mathcal{J})$).

Para cada nó intermediário de uma jornada, chamamos de tempo de espera o tempo entre a chegada ao nó e o instante de utilização da próxima aresta ($\sigma_i - \sigma_{i-1}$).

Podemos definir três tipos de jornadas ótimas em um grafo evolutivo, levando em conta medidas distintas para uma jornada $\mathcal{J} = (R, R_\sigma)$:

- O instante de chegada da mensagem ($arrival(\mathcal{J})$). Chamamos de *Foremost Journey* a jornada que chega no nó destino mais cedo. Definimos $a(s, r)$ como o tempo de chegada de uma jornada *foremost* de s a r.

- O número de arcos percorridos ($|R|$). A jornada que percorre o menor número possível de arcos entre a origem e o destino é chamada *Shortest Journey*. Definimos $d(s, r)$ como o número de saltos de uma jornada *shortest* de s a r.

- O tempo de percurso ($transit(\mathcal{J})$), ou seja, a diferença entre o instante de envio ($departure(\mathcal{J})$) e o instante de chegada ($arrival(\mathcal{J})$). A jornada com menor tempo de percurso é chamada *Fastest Journey*. Definimos $tt(s, r)$ como o tempo de trânsito de uma jornada *fastest* de s a r.

3.3 Estrutura de Dados e Implementação

A estrutura utilizada na representação de Grafos Evolutivos consiste em um vetor de listas de adjacência, sendo que, para cada vizinho de um nó, guarda-se o horário de atividade da aresta. Esta estrutura ocupa espaço proporcional a $N + M\delta_E$.

É conveniente para a implementação e análise dos algoritmos para cálculo de jornadas definir $f(e, t)$, com $e \in E_G$ e $t \in [1, \tau]$, como a função que, dada uma aresta e um instante de tempo, devolve o próximo instante após t no qual tal aresta pode ser atravessada (min $\{t' \mid t' \in P(e) \text{ e } t' \geq t\}$) ou ∞, caso este não exista.

Utilizando busca binária nos intervalos de uma aresta, conseguimos calcular o valor de $f(e, t)$ em tempo $O(\log \delta_E)$

4 Jornada Foremost

Para computar as jornadas com menor instante de chegada de um nó s a todos os outros em V, procedemos de forma parecida com o algoritmo de Dijkstra para cálculo de caminhos de menor custo em grafos comuns. A construção deste algoritmo depende do fato de que caminhos prefixos de caminhos de custo mínimo também são de custo mínimo. Este fato não é necessariamente verdade para jornadas *foremost* em grafos evolutivos, porém, a partir de qualquer uma destas jornadas, podemos construir uma nova que respeita a propriedade dos prefixos.

De fato, seja $\mathcal{J}$ uma jornada *foremost* de s a r em um grafo evolutivo. Seja u um nó intermediário desta jornada e seja $\mathcal{J}'$ uma jornada de s a u. Suponha que $\mathcal{J}'$ não é uma jornada *foremost* de s a u. Então, existe uma outra jornada $\mathcal{J}''$ de s a u que tem data de chegada mínima. Se trocarmos $\mathcal{J}'$ por $\mathcal{J}''$ como prefixo de $\mathcal{J}$, esta continua válida (pois o tempo de chegada de $\mathcal{J}''$ é menor que o de $\mathcal{J}'$ e, portanto, não influi no resto da jornada) e continua tendo tempo mínimo de chegada. Podemos repetir este processo para todos os prefixos que não sejam jornadas *foremost*.

Com isto, o cálculo da jornada *foremost* é realizado da seguinte maneira: construímos um conjunto C de vértices para os quais a jornada ótima já foi calculada (chamamos estes vértices de *fechados*) e um conjunto Q de vértices já visitados, mas cuja jornada ainda não foi calculada (chamamos estes vértices de *abertos*). A cada passo, escolhemos um vértice u, em Q, cuja estimativa de data de chegada é mínima e o inserimos em C. Em seguida, removemos u de Q, inserimos em Q todos os vizinhos de u que não estão lá ainda e atualizamos suas estimativas de data de chegada. Quando Q estiver vazio, significa que todos os vértices de V alcançáveis a partir do vértice inicial estão em C, ou seja, calculamos a jornada *foremost* para todos eles.

A seguir, apresentamos o algoritmo proposto em [XFJ02]. Neste algoritmo, Q é uma fila de prioridade implementada com um *min-heap*, $t_{foremost}$ e *pai* são um vetores que guardam, para cada nó, respectivamente, o menor instante de tempo no qual é possível chegar neste nó e seu pai em uma jornada *foremost*.

JORNADA-FOREMOST$(\mathcal{G}, s)$

```
1  t_foremost[s] ← 0
2  para v ≠ s, v ∈ V_G
3        faça t_foremost[v] ← ∞
4  Q ← {s}
5  enquanto Q ≠ ∅
6        faça u ← remove_raiz(Q) ▷ agora u está fechado
7              para cada vizinho aberto v de u
8                    faça t ← f((u, v), t_foremost[u])
9                          se t + ζ(u, v) < t_foremost[v]
10                            então t_foremost[v] ← t + ζ(u, v)
11                                  pai[v] ← u
12                                  Q ← Q ∪ {v}
13 devolva Jornadas determinadas por pai e t_foremost
```

A cada passo do laço da linha 5, um vértice é fechado, e nunca reinserimos um vértice fechado na fila de prioridade Q.

Vamos mostrar que, ao final do algoritmo, $t_{foremost}[u] = a(s, u)$ para todo vértice u de $\mathcal{G}$ por indução no conjunto C de vértices fechados. Inicialmente, temos $C = \{s\}$ e $t_{foremost}[s] = 0 = a(s, s)$.

Suponha agora que, em alguma iteração do algoritmo, o conjunto C foi corretamente computado ($t_{foremost}[w] = a(s, w)$ para todo vértice $w \in C$) e um vértice v está prestes a ser fechado. Portanto, v foi inserido no *heap* Q, logo, v é alcançável por s. Seja $\mathcal{J}$ uma jornada *foremost* de s a v, cujos prefixos também são jornadas *foremost*. Esta jornada liga o vértice s dentro de C com o vértice v fora de C. Seja agora v' o primeiro vértice em $\mathcal{J}$ fora de C e seja u o vértice que precede v' imediatamente em $\mathcal{J}$, ou seja, $u \in C$.

Como C foi corretamente computado, temos que $t_{foremost}[u] = a(s, u)$. Quando u foi fechado, v' foi inserido em Q e, como v' vem antes de v em $\mathcal{J}$, $t_{foremost}[v'] \leq t_{foremost}[v]$ e, como v é escolhido pelo algoritmo como o vértice com menor $t_{foremost}$ fora de C, $v' = v$ e $t_{foremost}[v'] = t_{foremost}[v]$. Além disso, $a(s, v) = f((u, v), a(s, u)) + \zeta(u, v)$, portanto, $t_{foremost}[v] = a(s, v)$.

O laço da linha 5 é executado $\Theta(N)$ vezes (pois cada vértice aparece apenas uma vez na fila). Sabemos que a operação *remove_raiz* do *heap* Q leva tempo $O(\log N)$ e esta é executada N vezes. Além disso, ao examinar todos os vizinhos de cada nó, no laço da linha 7, estamos percorrendo todas as M arestas do grafo e, para cada uma, calculamos a função f, que tem complexidade $O(\log \delta_E)$.

Portanto, podemos concluir que o algoritmo JORNADA-FOREMOST calcula as jornadas com menor tempo de chegada de um nó s para todos os outros nós em tempo $O(M \log \delta_E + N \log N)$.

5 Jornada Shortest

Calcular a jornada com número mínimo de arestas é uma tarefa mais complexa que a anterior devido ao fato de o prefixo de uma jornada ótima não ser necessariamente uma jornada ótima.

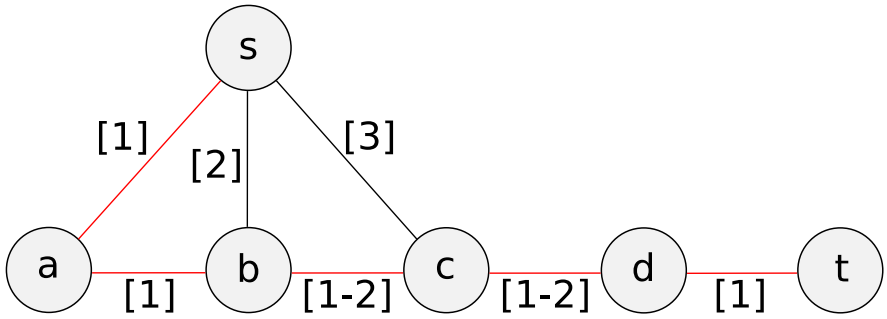


Figura 4: A jornada mais curta de s a t é a que passa por todos os nós intermediários no instante 1.

Isso acontece quando temos uma jornada *shortest* $\mathcal{J}$ de s a r e uma aresta intermediária (u, v) que só pode ser percorrida até o instante t. Neste caso, se houver uma jornada $\mathcal{J}'$ de s até u que seja mais curta do que a já encontrada, mas que chegue no nó u após o instante t, não podemos trocar o prefixo de $\mathcal{J}$ por $\mathcal{J}'$, pois a aresta (u, v) não poderia ser percorrida.

No entanto, podemos notar que, se um prefixo de uma jornada *shortest* ($s \rightarrow u$, por exemplo) termina em um instante t, tal prefixo é a jornada mais curta de s até u que termina antes de t (ou poderíamos trocar o prefixo, obtendo uma jornada válida mais curta).

Para calcular as jornadas shortest de um nó s, calculamos a cada iteração, os caminhos de menor tempo de chegada com no máximo k saltos, a partir dos caminhos de tamanho $k - 1$. Fazendo isso, acabamos obtendo uma árvore de prefixos em que cada nó pode aparecer mais de uma vez, mas sem duplicatas no mesmo nível da árvore.

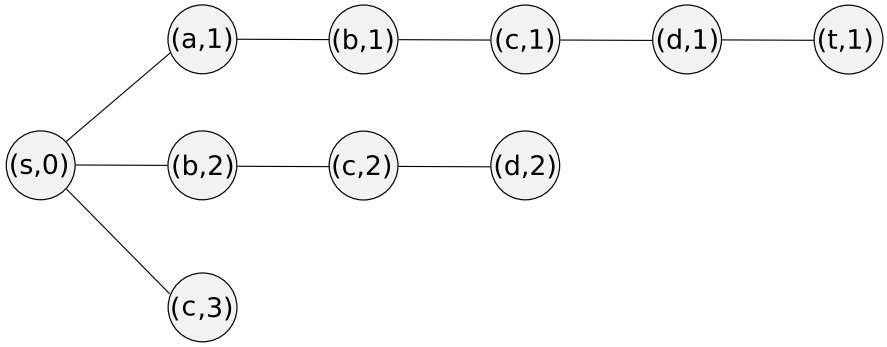


Figura 5: Árvore de predecessores correspondente ao grafo da Figura 4.

O algoritmo de cálculo de jornada *shortest* utiliza a subrotina SELECAO-DE-ARESTAS-E-HORÁRIOS para calcular, a partir do nível k, quais arestas e horários fazem parte do nível $k+1$ da árvore de predecessores. Este processo é feito a partir dos tempos mínimos de chegada em cada nó do grafo. Com esta informação, o algoritmo verifica, em cada aresta, se é possível chegar na ponta final em um instante mais cedo do que o já atingido. Todas as arestas em que existe esta possibilidade são guardadas, bem como o tempo mínimo de travessia.

SELECAO-DE-ARESTAS-E-HORÁRIOS($\mathcal{G}, t_{LBD}$)

```
1  para v ∈ V_G
2        faça e_min[v] ← nil
3             t_min[v] ← ∞
4             t_chegada[v] ← t_LBD[v]
5  para (u, v) ∈ E_G
6        faça t ← f((u, v), t_LBD[u])
7             se t + ζ(u, v) < t_chegada[v]
8                então e_min[v] ← (u, v)
9                     t_min[v] ← t
10                    t_chegada[v] ← t + ζ(u, v)
11 devolva (e_min, t_min)
```

No pseudo-código acima, $t_{LBD}[u]$ representa o menor tempo em que é possível sair do nó u (*Lower Bound on Departure*), $e_{min}[u]$ e $t_{min}[u]$ guardam, respectivamente, a primeira aresta que se pode utilizar para sair de u após $t_{LBD}[u]$ e seu tempo de transição.

Como este algoritmo percorre todas as arestas do grafo e, para cada uma, calcula a função f, sua complexidade é $O(M \log \delta_E)$.

Desta forma, para calcular as jornadas mais curtas, deve-se chamar a subrotina SELECAO-DE-ARESTAS-E-HORÁRIOS para cada nível k da árvore de predecessores. Com as arestas que pertencem ao nível atual, deve-se verificar se o nó final da aresta já foi alcançado antes. Em caso negativo, a jornada que utiliza esta aresta é a mais curta possível para aquele nó e, em caso afirmativo, a jornada mais curta foi encontrada anteriormente, mas, esta jornada deve ser guardada, pois ainda pode ser prefixo de alguma jornada *shortest* para outro nó.

Como somente atualizamos a jornada até um nó intermediário se esta tem instante de chegada menor que a jornada anterior, ao final da iteração k, o algoritmo mantém uma lista com todas as jornadas *foremost* com até k saltos de s a todos os nós já alcançados.

```
JORNADA-SHORTEST(G, s)
 1  para v ∈ V_G
 2        faça t_LBD[v] ← ∞
 3             J[v] ← ∅
 4  t_LBD[s] ← 0
 5  J_shortest[s] ← ∅
 6  k ← 1
 7  inalcancaveis ← N − 1
 8  enquanto inalcancaveis > 0 e k < N
 9        faça k ← k + 1
10             (e_min, t_min) ← SELECAO-DE-ARESTAS-E-HORÁRIOS(G, t_LBD)
11             se SELECAO-DE-ARESTAS-E-HORÁRIOS não alterou nada
12                então devolva J_shortest
13             para v ∈ V_G tal que e_min[v] ≠ nil
14                  faça Seja (u, v) = e_min[v]
15                       Seja (R, R_σ) = J[u]
16                       J[v] ← (R ∪ e_min[v], R_σ ∪ t_min[v])
17                       se t_LBD[v] = ∞
18                          então J_shortest[v] ← J[v]
19                                inalcancaveis ← inalcancaveis −1
20                       t_LBD[v] ← t_min[v] + ζ(e_min[v])
21  devolva J_shortest
```

O laço principal do programa executa no máximo $N - 1$ vezes. No interior deste laço, é executada uma chamada do algoritmo SELECAO-DE-ARESTAS-E-HORÁRIOS, cuja complexidade é $O(M \log \delta_E)$, bem como um outro laço com consumo de tempo $O(N)$. Logo, a complexidade total do algoritmo é $O(N(M \log \delta_E + N))$.

Como vimos na seção anterior, sempre podemos construir uma jornada *foremost* de k saltos adicionando uma aresta a alguma jornada *foremost* de $k - 1$ saltos. Como a única jornada de tamanho 0 é a de s a s, concluímos que o algoritmo JORNADA-SHORTEST encontra, a cada iteração, todas as jornadas de menor número de saltos com no máximo k arestas.

6 Jornada Fastest

Assim como na jornada *shortest*, o prefixo de uma jornada *fastest* não é, necessariamente, uma jornada ótima. De fato, calcular a jornada de menor tempo de trânsito é ainda mais difícil do que calcular a mais curta.

A figura 6 mostra a diferença entre as jornadas *fastest* para cada nó. Arestas de uma mesma cor representam as jornadas ótimas para os vértices correspondentes.

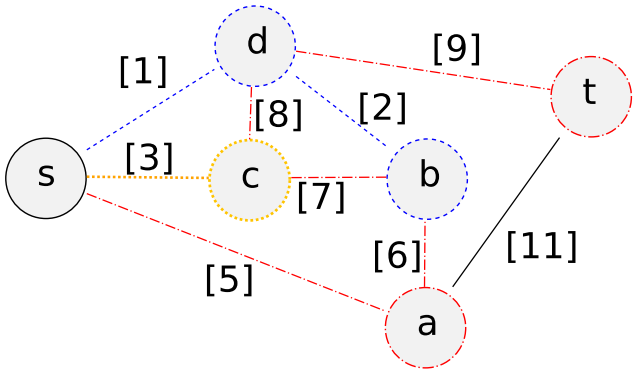


Figura 6: Jornadas *fastest* a partir de s em um grafo evolutivo.

Podemos perceber que uma jornada *fastest* com tempo de partida t é sempre uma jornada *foremost* dentre as iniciadas a partir de t.

De fato, seja P uma jornada *fastest*, e seja $t = saida(P)$. Suponha que a afirmação seja falsa. Então há uma jornada *foremost* P^* iniciada a partir de t e com tempo de chegada $< chegada(P)$. Obviamente, $saida(P^*) \geq t$ (pois a jornada é iniciada a partir de t). Assim, o tempo de trânsito dessa mensagem será $chegada(P^*) - saida(P^*) < chegada(P) - saida(P)$ e, portanto, P não é *fastest*.

A ideia do algoritmo proposto por [Jar05] é, portanto, calcular a jornada *foremost* a partir de todos os tempos "relevantes", ou seja, os instantes em que a jornada *foremost* para algum nó pode mudar.

```
JORNADA-FASTEST(G, s)
 1  t_d ← 0
 2  para cada vértice x
 3        faça t(s, x) ← +∞
 4  enquanto t_d < +∞
 5        faça Δ ← +∞
 6              J ← JornadaForemost(G, s, t_d)
 7              para cada vértice v
 8                    faça J_v ← atrase(J_v)
 9                          atraso = departure(J_v) − t_d
10                          se t(s, x) > transittime(J_v)
11                             então t(s, x) ← transittime(J_v)
12                                   t_d(s, x) ← departure(J_v)
13                          Seja t' o primeiro instante em que um arco de J_v deixa
                            deixa de existir.
14                          Δ ← min(Δ, (t' − arrival(J_v) + atraso))
15              t_d ← t_d + Δ
16  para cada vértice x
17        faça J(x) ← JORNADA-FOREMOST(G, s, t_d(s, x))
18  devolva J
```

A cada iteração do laço da linha 4, as jornadas *foremost* a partir de t_d são computadas e, para cada nó, é calculado o maior atraso possível que não afeta o resultado. Então, sendo Δ o menor desses atrasos, começamos a nova iteração adicionando Δ a t_d.

Observe que muitas vezes é possível 'atrasar' ao máximo a jornada mantendo o tempo de chegada fixo e diminuindo o tempo de trânsito. A figura 7 mostra um exemplo disso. Podemos perceber que o tempo ganho com tal atraso pode ser adicionado a t_d sem afetar o resultado para este nó, já que sabemos que, para qualquer mensagem enviada a partir de t_d, não há jornada que chegue antes de $a(\mathcal{J})$.

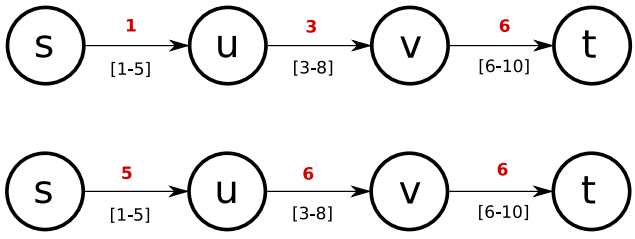


Figura 7: Acima, uma jornada *foremost* ($s \xrightarrow{1} u \xrightarrow{3} v \xrightarrow{6} t$). Abaixo, a jornada atrasada, mantendo o tempo de chegada ($s \xrightarrow{5} u \xrightarrow{6} v \xrightarrow{6} t$)

A única forma de a árvore de predecessores criada pelo algoritmo JORNADA-FOREMOST mudar, a menos de atraso na rota, entre um instante e outro é se uma das arestas da árvore desaparecer. É fácil ver que, para haver tal mudança, alguma aresta do grafo deve aparecer

ou desaparecer. Porém, se uma aresta que aparece cria uma jornada que chega antes sem invalidar a encontrada anteriormente, essa nova jornada já era *foremost* no instante anterior. Portanto só precisamos considerar as desconexões.

Considere, então, que em uma iteração do algoritmo foram calculadas as jornadas *foremost* $\mathcal{J}$ a partir de um instante t_d. Seja (u_v, x) a primeira aresta que desaparece da jornada de s a v e seja t' o instante em que essa aresta deixa de existir. Como (u_v, x) é a primeira aresta a desaparecer, sabemos que, no intervalo $[t_d, t']$, a jornada *foremost* de s a u_v está disponível e pode ser atrasada até que não haja tempo de espera em nenhum nó interno (Como as arestas existem no intervalo, podemos atrasar a aresta anterior a u até não haver tempo de espera, depois atrasar a anterior a essa e assim por diante).

Assim, a jornada *foremost* de s a v permanecerá a mesma no intervalo $[t_d, t' - \mathit{transit}(\mathcal{J}_{u_v})[$. Seja $\epsilon_v = t' - \mathit{transit}(\mathcal{J}_{u_v}) - t_d$ calculado conforme descrito acima, e seja $\Delta = min(\epsilon_v : v \in V_\mathcal{G})$. Podemos observar que as jornadas encontradas pelo algoritmo durante o intervalo $[t_d, t_d + \Delta[$ seriam equivalentes e podemos pular este intervalo.

Então, como todas as jornadas *fastest* são *foremost* para algum instante, e o algoritmo calcula todas as jornadas *foremost* relevantes, concluímos que a jornada encontrada ao fim do algoritmo é a de menor tempo de trânsito.

O número de instantes relevantes será $O(\mathcal{M})$ (o número de desconexões). *atrase* consome tempo $O(N)$. Cada iteração do laço externo consome o tempo de uma execução do algoritmo *foremost* mais N^2.

Portanto, o consumo de tempo do algoritmo Jornada-Fastest é $O(\mathcal{M}(M \log \delta_E + N^2))$.

7 Experimentos

Os algoritmos descritos acima foram implementados em Java. O programa recebe arquivos com os horários em que as conexões da rede ocorrem e em que as mensagens são criadas e calcula a jornada a partir do nó desejado.

Para testar a implementação, foi utilizado um simulador de contextos reais de DTNs, no qual o roteamento das mensagens foi feito com as jornadas ótimas calculadas. O simulador escolhido foi o *Opportunistic Network Environment* (ONE), pois este permite a fácil utilização de estratégias de roteamento criadas pelo usuário e permite simulações nas quais os nós se movem por um mapa para certos pontos de interesse, em vez de apenas se moverem aleatoriamente pelo plano. Além disso, o ONE apresenta grande facilidade para a obtenção de dados das simulações, através de relatórios claros e bastante variados.

Os modelos de movimentação utilizados foram o baseado em mapas, com o mapa da cidade de Helsinque, e o *Random Way Point* (RWP). No primeiro modelo, cada nó escolhe um ponto do mapa como seu destino e encontra um caminho (utilizando as vias disponíveis) até ele. No segundo, cada nó simplesmente escolhe um ponto no plano e se dirige a ele em linha reta.

Fazer o ONE utilizar rotas ótimas em suas simulações não foi uma tarefa simples, pois, nele, não existe nenhum conhecimento prévio sobre a rede. Para contornar este problema, utilizamos os relatórios gerados pelas simulações com os dados de início e fim de conexões entre nós e tempo e local de criação de mensagens. A partir da informação sobre as conexões, foi possível a construção do grafo evolutivo correspondente à simulação e, com a informação sobre a criação de mensagens, foi possível determinar o local e tempo de início das rotas a serem calculadas. Com isto em vista, foi acoplado ao programa que calcula as jornadas ótimas um interpretador que recebe os dados do ONE e gera os grafos evolutivos necessários e constrói a classe de roteamento do ONE.

Como o ONE não permite a entrega de duas mensagens em uma mesma conexão ao mesmo tempo, processamos as mensagens em ordem, e, quando uma rota é encontrada, as arestas utilizadas são removidas do grafo (nos instantes pertinentes) para o cálculo das próximas jornadas.

Para passar o roteamento encontrado de volta para o ONE, foi criada uma classe de roteamento com uma tabela de hash associando o identificador de uma mensagem com o instante de tempo e o destino de cada salto que esta deve realizar para ser entregue com sucesso. Deste modo, para cada simulação que desejarmos fazer com as rotas ótimas, devemos:

- Rodar o ONE uma vez para obter as informações de conectividade e criação de mensagens.

- Com base nas informações, rodar os algoritmos e calcular as jornadas, gerando a classe de roteamento para o ONE.
- Rodar o ONE novamente, desta vez com a classe de roteamento criada com as rotas ótimas.

Após rodar a simulação utilizando as jornadas calculadas, o ONE gera relatórios com os dados das mesmas. Utilizamos um script em Perl para realizar todas estas operações com o ONE e extrair os dados dos relatórios obtidos para que fosse possível a análise dos resultados.

8 Análise dos Resultados e Conclusões

Um dos pontos mais importantes que desejávamos verificar era o funcionamento dos algoritmos implementados. Para isto, variamos o número de nós na rede (aumentando, consequentemente, o número de nós e arestas do grafo evolutivo correspondente) e extraímos das jornadas utilizadas nos roteamentos do ONE os dados que os algoritmos implementados deveriam otimizar: a latência (diferença entre o tempo de chegada da mensagem e seu tempo de criação), o número de saltos (arestas percorridas) e o atraso (diferença entre o tempo de chegada da mensagem e o tempo de sua primeira retransmissão).

Para realizar as simulações necessárias, definimos um cenário padrão. Fixamos o raio de transmissão dos nós em 15m, sua velocidade entre 20 e 100m/s, a velocidade de transmissão de mensagens em 250kb/s, o tamanho das mensagens em 5kb e o tamanho do *buffer* dos nós em 100kb. A velocidade de transmissão é muito maior que o tamanho das mensagens para que cada transmissão demore exatamente um segundo. Além disso, cada simulação dura 3000s e o intervalo entre a criação de mensagens é entre 25 e 35s. O padrão de movimentação utilizado foi o movimento baseado em caminhos mais curtos no mapa de Helsinque.

Para que os dados obtidos tivessem certa relevância estatística, repetimos o experimento dez vezes para cada ponto de cada curva, utilizando uma semente diferente para os números aleatórios em cada simulação. Tomamos a média, o valor mínimo e o valor máximo para cada um dos valores e os plotamos, obtendo as curvas abaixo.

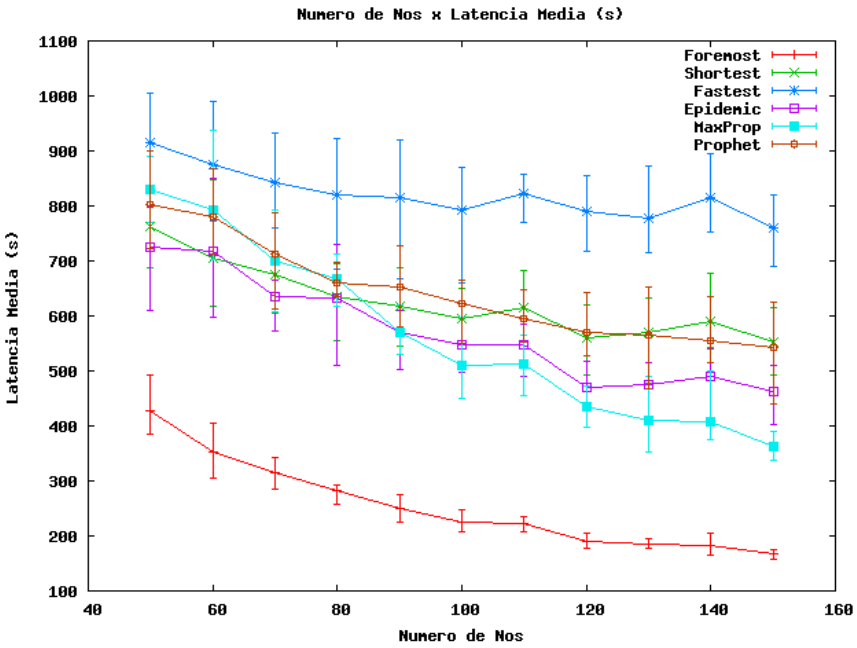


Figura 8: Número de nós x Latência Média

No gráfico 8, podemos verificar que o algoritmo *foremost* realmente possui a menor latência dentre os roteamentos estudados. Além deste fato, vemos que o protocolo MaxProp apresenta resultados piores que os outros quando a rede é pequena, mas, à medida que o número de nós cresce, sua latência diminui consideravalmente. O algoritmo *fastest* apresenta a pior latência de todas, o que pode ser explicado pela espera que este tipo de jornada realiza no nó inicial para atingir o menor tempo de trânsito possível.

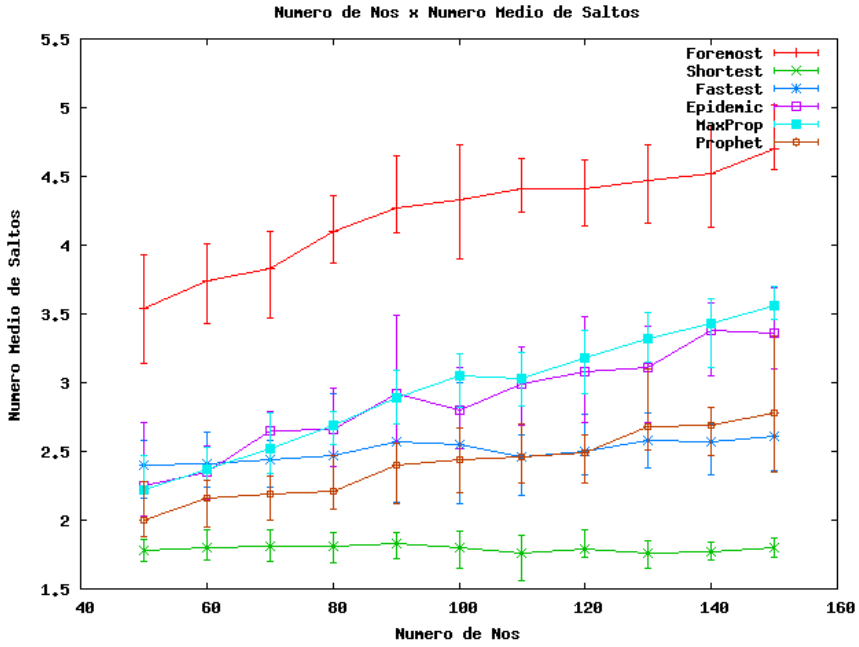


Figura 9: Número de nós x Número médio de saltos por mensagem

O gráfico 9 nos mostra que, na prática, a jornada *shortest* realmente realiza o menor número possível de saltos. Outro fato interessante a ser notado é que a jornada *foremost* passa por muito mais arestas que os outros protocolos. Isso acontece pois a jornada que chega mais cedo não pode esperar por ligações diretas entre o destino e a origem, por isso, toma rotas alternativas que passam por mais nós.

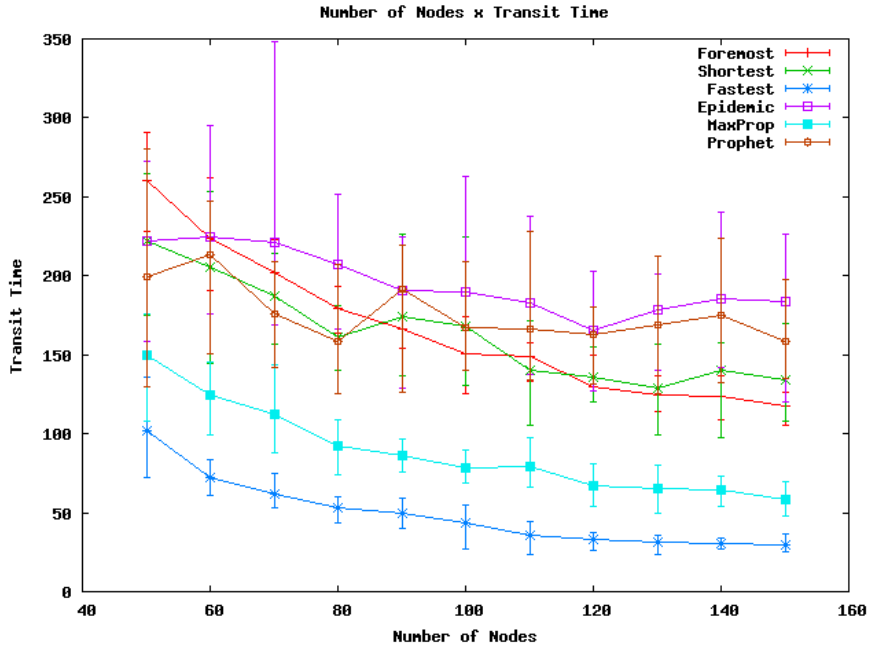


Figura 10: Número de nós x Tempo médio de trânsito por mensagem

O gráfico 10 certifica que o algoritmo *fastest* possui o menor entre os atrasos. Além disso, percebemos que este parâmetro não segue um padrão facilmente discernível nos algoritmos vistos. Os valores sobem e descem quase aleatoriamente conforme a rede cresce. A única exceção a este fato é o protocolo MaxProp, que, além de apresentar baixo tempo de trânsito nas rotas, sua distribuição segue um padrão linear. Isso nos mostra que este protocolo é o mais indicado se estivermos interessados em obter rotas que priorizem este parâmetro.

Além das conclusões óbvias, podemos notar nestes gráficos que os três parâmetros que tentamos otimizar não possuem nenhuma relação direta entre eles. Ou seja, otimizar um deles não garante nenhuma propriedade sobre os outros.

Outro objetivo que buscamos ao desenvolver o roteamento com conhecimento prévio sobre a rede é obter probabilidades mais altas de entrega e menos carga da rede. A partir das simulações, também obtivemos estes dados.

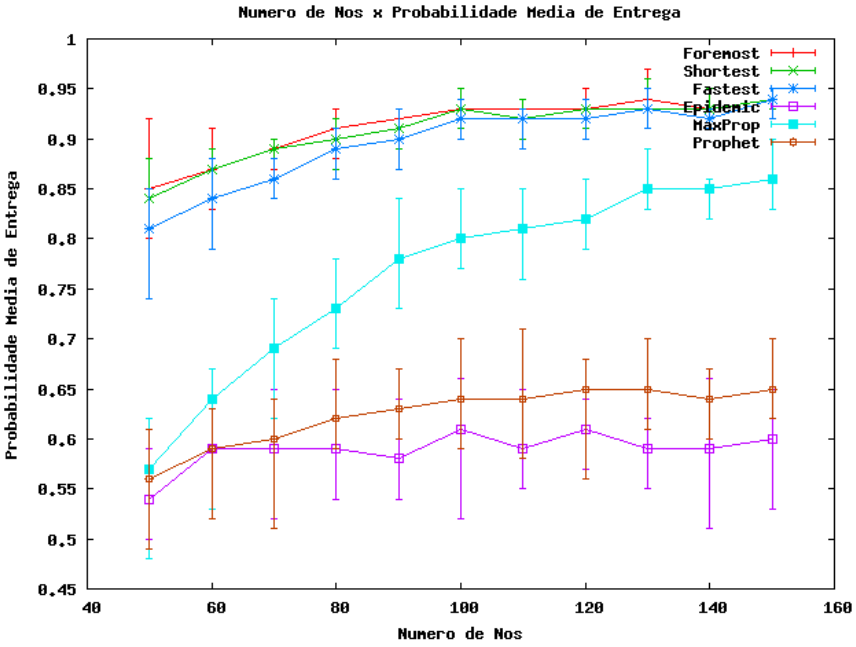


Figura 11: Número de nós x Probabilidade de entrega

O gráfico 11 mostra que a entrega de mensagens dos algoritmos com conhecimento sobre a rede é consideravelmente maior, mesmo comparada com o protocolo com melhores resultados.

No entanto, apesar da alta taxa de entrega, esta não é necessariamente ótima, uma vez que apenas uma mensagem pode ser transmitida de cada vez e, portanto, uma mensagem pode 'bloquear' uma aresta que é necessária para uma mensagem posterior, sendo que ambas poderiam ser entregues caso fosse escolhida uma outra rota para a primeira.

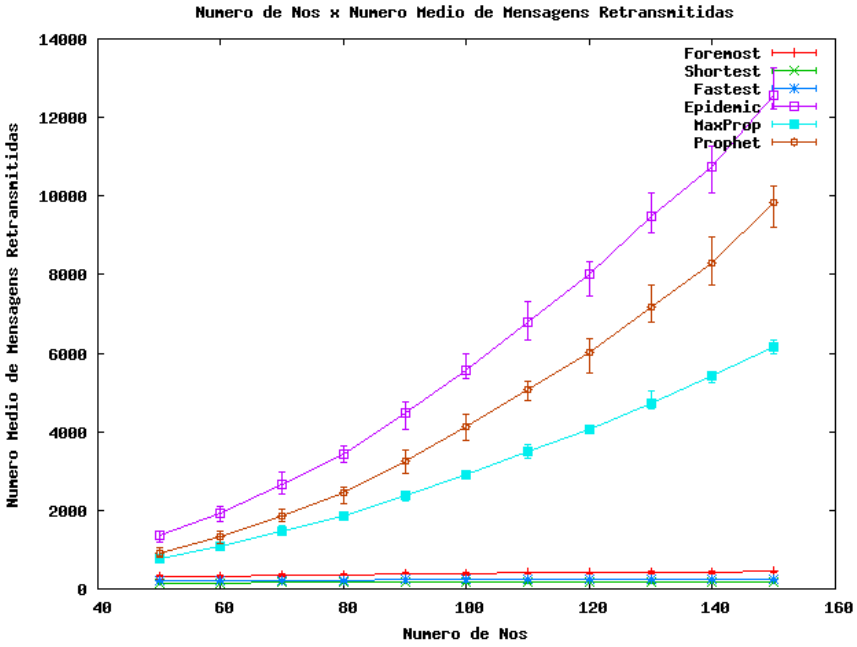


Figura 12: Número de Nós x Número de mensagens Retransmitidas

Podemos perceber, também, pelo gráfico 12 que as rotas que utilizam jornadas pré-calculadas realizam muito menos transmissões por mensagem. Este fato é esperado, pois cada mensagem só é retransmitida ao longo da rota correta, o que não ocorre com os outros algoritmos, que podem replicar os pacotes para aumentar a probabilidade de entrega.

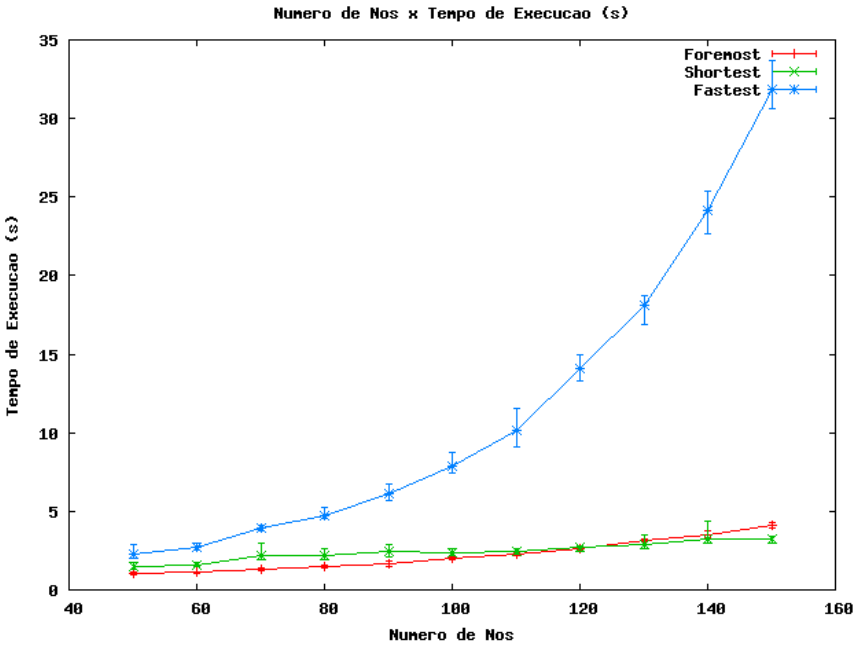


Figura 13: Número de nós x Tempo de execução

O gráfico da figura 13 mostra o tempo de execução de cada algoritmo em relação ao número de nós da simulação. Podemos perceber que, apesar da análise de complexidade mostrar que no pior caso o algoritmo *shortest* é mais lento que o *foremost*, na prática ambos levam, em média, o mesmo tempo. Devido ao uso repetido do algoritmo *foremost* é fácil ver que o algoritmo *fastest* consome muito mais tempo do que os outros.

Considerando todos os resultados apresentados neste trabalho, podemos perceber que conhecer a topologia de uma DTN pode fazer uma grande diferença na hora de encaminhar as mensagens. Não só podemos otimizar a rota utilizada, mas também diminuímos consideravelmente a carga da rede e dos *buffers* dos nós e ainda aumentamos a proporção de mensagens entregues.

Apesar disso, como nem sempre uma DTN pode ser previsível, o uso dos roteamentos ótimos pode servir como um limite inferior na comparação os protocolos probabilísticos, mais indicados para redes dinâmicas em geral. Tal abordagem pode ser útil na criação de novos algoritmos de roteamento no futuro.

Referências

[BGJL06] J. Burgess, B. Gallagher, D. Jensen, and B.N. Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *Proc. IEEE Infocom*, pages 1–11, 2006.

[Fer02] A. Ferreira. On models and algorithms for dynamic communication networks: the case for evolving graphs. In *In Proc. ALGOTEL*, 2002.

[Jar05] Aubin Jarry. *Connexité dans les réseaux de télécommunications*. PhD thesis, Université de Nice Sophia Antipolis, FR, 2005.

[KOK09] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *SIMUTools '09: Proceeding of the 2nd International Conference on Simulation Tools and Techniques*, New York, NY, USA, 2009. ICST.

[LDS04] A. Lindgren, A. Doria, and O. Schelen. Probabilistic routing in intermittently connected networks. *Lecture Notes in Computer Science*, pages 239–254, 2004.

[MGF07] J. Monteiro, A. Goldman, and A. Ferreira. Using Evolving Graphs Foremost Journey to Evaluate Ad-Hoc Routing Protocols. In *In Procedings of 25th Brazilian Symposium on Computer Networks (SBRC'07), Belem, Brazil*, 2007.

[OMR+07] C.T. Oliveira, M.D.D. Moreira, M.G. Rubinstein, L. Costa, and O. Duarte. Redes tolerantes a atrasos e desconexoes. In *Minicursos do Simposio Brasileiro de Redes de Computadores (SBRC 2007)*, 2007.

[SPR05] T. Spyropoulos, K. Psounis, and C.S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259. ACM New York, NY, USA, 2005.

[VB00] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical report, Citeseer, 2000.

[XFJ02] B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. 2002.

Parte II

Parte Subjetiva: César

Fiquei sabendo deste projeto no fim de 2008 e fiquei bem interessado (na época, não tinha ideia do que fazer para o TCC).

Durante os primeiros meses, nós cinco estudamos coisas mais gerais sobre DTNs e grafos evolutivos e tentávamos decidir quais temas seriam usados para as monografias.

No fim do primeiro semestre de 2009, chegamos aos três temas e eu e o Paulo passamos a nos concentrar mais na implementação e análise dos algoritmos (na época apenas o foremost já tinha sido concluído).

Tivemos dificuldades com o algoritmo *fastest* que tinhamos encontrado e, após um exame mais detalhado, percebemos que haviam erros no pseudocódigo do artigo estudado. Tivemos, então, uma ideia diferente para o cálculo da jornada *fastest* e começamos a implementá-la, mas tivemos alguns problemas com o consumo de memória do algoritmo.

Nesse momento tivemos que parar temporariamente o desenvolvimento dos algoritmos para fazer com que estes funcionassem em conjunto com o simulador ONE. Esta tarefa foi mais difícil do que imaginávamos porque, sendo um simulador de roteamento, o ONE não permitia o cálculo prévio de rotas.

Após conseguirmos rodar as simulações, voltamos a investigar o algoritmo *fastest* e descobrimos que o autor do artigo original havia publicado uma versão diferente (e bem mais simples) do algoritmo em sua tese de doutorado (em francês).

Apesar de o algoritmo estar correto e ser bem mais simples, sua implementação foi complicada pois em alguns lugares o pseudocódigo era muito vago.

Escrevendo a prova do algoritmo para esta monografia, percebemos que parte dos cálculos feitos era desnecessária e conseguimos melhorar a implementação.

Uma vez implementados e simulados os três algoritmos, voltamos a pensar na outra ideia para o *fastest*. Após algumas tentativas sem sucesso, conseguimos criar uma maneira de evitar o estouro da memória, mas não tivemos tempo de implementá-la ainda.

Pretendemos comparar esse novo algoritmo com o anterior e escrever um artigo sobre isso para o Simpósio Brasileiro de Redes de Computadores.

1 Disciplinas Mais Relevantes

- MAC0110 Introdução à Computação

- MAC0122 Princípios de Desenvolvimento de Algoritmos
  Estas matérias introdutórias me ensinaram a programar.

- MAC0323 Estruturas de Dados
  Durante o projeto modificamos as estruturas de alguns dos algoritmos para facilitar a implementação.

- MAC0328 Algoritmos em Grafos

- MAC0325 Otimização Combinatória

- MAC0330 Teoria dos Grafos
  O projeto foi completamente baseado em teoria dos grafos e estas matérias foram muito úteis para o entendimento dos algoritmos estudados.

- MAC0338 Análise de Algoritmos
  Disciplina de grande utilidade para analisar a complexidade dos algoritmos, também serviu para percebermos que um dos algoritmos fazia cálculos desnecessários.

Parte III

Parte Subjetiva: Paulo

1 Como tudo começou

Antes de ingressar no grupo de estudos sobre DTNs, eu estava bastante incerto sobre o que faria no trabalho de conclusão. Até cogitei conversar com alguns professores, mas não tinha nenhuma ideia de que tipo de projeto eu gostaria de trabalhar.

No final de 2008, porém, fui convidado a participar de um grupo de estudos coordenado pelo professor Alfredo e comecei a frequentar as reuniões. Inicialmente, as discussões eram bastante vagas. Líamos sobre DTNs e conversávamos sobre aplicações, trabalhos já feitos e possíveis trabalhos futuros da área.

Então, chegou 2009 e começamos a levar o trabalho mais a sério. O professor Alfredo já havia trabalhado com a modelagem de DTNs com grafos evolutivos anteriormente e nos sugeriu o tema. Como tanto eu quanto o César gostamos da área de combinatória e grafos, ficamos mais que felizes em trabalhar com esta extensão da teoria dos grafos.

2 Desenvolvimento do Projeto: Desafios e Frustrações

Logo no início do ano, começamos a implementar os algoritmos do zero, em um sistema *Java* criado por nós mesmos. Escolhemos esta linguagem por oferecer estruturas de dados mais facilmente acessíveis, como filas de prioridade e listas, que são muito utilizadas na implementação de algoritmos para grafos.

O algoritmo *foremost* foi o primeiro a ser implementado e não ofereceu muita dificuldade. Inicialmente, nossa implementação era bem parecida com o pseudocódigo, que devolvia uma rede de predecessores e um vetor de tempos. Esta era bastante simples, mas não ficou muito bonita. O próximo algoritmo implementado foi o *shortest*, que demandou um pouco mais de tempo, devido a algumas sutilezas na hora de guardar as jornadas. Novamente, a implementação não estava exatamente como queríamos.

Depois deste "sprint" inicial, acabamos ficando cerca de um mês com o trabalho parado, tentando entender o algoritmo *fastest*. A versão que está no artigo de onde tiramos o *foremost* e o *shortest* utiliza uma outra ideia e é muito mais complicada. Muito tempo depois descobriríamos que havia alguns erros naquele pseudocódigo. Tentando contornar este problema, acabamos chegando a uma ideia diferente para calcular a jornada *fastest*. Este novo algoritmo tinha uma ideia diferente, que parecia promissora e ficamos bastante empolgados com ele.

Era relativamente fácil a implementação, então fizemos isso rapidamente, mas quando fomos rodar, a máquina travou, pois ele consumia muita memória. Isto fez com que deixássemos esta ideia um pouco de lado.

Neste meio tempo, o professor Alfredo teve um artigo com este tema aceito por uma conferência na Lapônia e precisava que fizessemos algumas simulações comparando os algoritmos com os protocolos de roteamento já existentes. Mas, o prazo para isto era bem curto e tivemos que parar o trabalho no *fastest* por um tempo. Precisávamos fazer toda a adaptação para o ONE e a implementação atual não permitia. Então, tivemos que refatorar os algoritmos para que eles devolvessem uma coleção de jornadas e, além disso, tivemos que implementar o *parser* para os relatórios do ONE, a classe de roteamento que utilizava as jornadas ótimas através de uma tabela de *hash* estática criada "na mão" (pois era a única maneira que conseguimos encontrar) e um *script* para fazer todas as simulações automaticamente e extrair os dados. Todas estas tarefas tiveram que ser feitas em apenas uma semana!

Felizmente, éramos cinco pessoas e ainda tivemos ajuda de um aluno do mestrado, então conseguimos terminar tudo a tempo da apresentação. Estas oscilações bruscas na quantidade de tarefas foram bastante frustrantes, pois acredito que poderíamos ter utilizado melhor o tempo.

Depois desta aventura, já estávamos no final de agosto e ainda não havíamos tido progresso com relação ao *fastest*. Quando o César foi analisar melhor o algoritmo, percebeu que haviam certos erros, então, resolvemos pesquisar os próximos trabalhos daqueles autores. Foi aí que encontramos a tese de doutorado com o algoritmo *fastest* que mostramos neste trabalho. O algoritmo era muito mais simples e não continha erros, mas, a tese estava toda escrita em francês. Apesar do trabalho de tradução, conseguimos implementar o algoritmo pouco tempo depois e adicioná-lo às simulações.

Assim, chegou outubro e começamos a preparar a monografia, o pôster e a apresentação. O projeto já estava praticamente pronto, então, bastava colocar tudo no papel. O algoritmo *fastest* proposto por nós ainda está em desenvolvimento, pois estamos tentando diminuir o consumo de memória sem alterar o funcionamento.

Trabalhar em um grupo de cinco pessoas foi uma experiência bastante interessante. Apesar de dividirmos os temas, todos nós continuávamos trabalhando juntos e nos reunindo semanalmente para discutir as novas ideias e o progresso do trabalho. Essa união proporcionou alguns desentendimentos, algumas reclamações, mas nada duradouro, pois, acima de tudo, estávamos trabalhando entre amigos e isso foi o mais importante para que o projeto desse certo.

3 Futuro

Atualmente, estamos escrevendo um artigo para o Simpósio Brasileiro em Redes de Computadores baseado neste trabalho. Se aceito, o apresentaremos em Gramado em maio de 2010.

Inicialmente, queríamos estudar a implementação de grafos evolutivos e dos algoritmos de cálculo de rotas de forma distribuída, mas não houve tempo. Eu pretendo seguir este caminho no mestrado e estudar outras modelagens para redes dinâmicas distribuídas.

4 Disciplinas Relevantes

- **Introdução à Computação, Principios de Desenvolvimento de Algoritmos e Estruturas de Dados** - As matérias introdutórias foram muito importantes por desenvolver meu gosto por programação e pela área de algoritmos.
- **Algoritmos em Grafos, Análise de Algoritmos e Otimização Combinatória** - Matérias da área de algoritmos muito importantes como embasamento teórico para o projeto e também matérias que eu gostei muito de fazer.
- **Desafios de Programação** - Apesar de consumir minha vida completamente por um semestre, me ajudou a melhorar meu jeito de programar e também a desenvolver meu gosto por competições de programação.