Controle de roteamento em redes $mesh$ via SDN usando heurísticas para "Unsplittable Flow Problem"

Henrique Duarte Moura[1], **Daniel Fernandes Macedo**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{henriquemoura,damacedo}@dcc.ufmg.br

Abstract. *Wireless networks are becoming more and more common. They can be used for infrastructure, since no cabling is required for the interconnection, especially $mesh$ networks. $Mesh$ networks pose challenges for network administrators due to the transmission medium's dynamics and peculiar characteristics. One challenge is maximizing the flow of the network using all links. With Software Defined Networks, where a controller makes the routing decisions, we can implement routing algorithms that consider the network throughput. In this way, routing can be reduced to solving a Unsplittable Flow Problem (UFP). UFP is a NP-hard problem, so we seek solutions based on heuristics and metaheuristics. We analyze constructive heuristics, such as Randomized Rounding, PROUTE, SPROUTE, and ESPROUTE, and develop a heuristic based on Dijkstra algorithm that provides higher flow rates for the tested instances, although it is slower than the others. We also evaluate a search heuristic proposed by Gamst, and we propose three variations, that show better results.*

Introdução

Redes sem fio estão se tornando cada vez mais comuns como meio de comunicação na sociedade moderna. Elas também são uma boa opção para montagem de infraestrutura, uma vez que não é necessário lançar cabeamento entre os pontos de transmissão. Redes $mesh$ [4] são usadas para esta finalidade. Existem várias qualidades deste paradigma, que incluem implantação de baixo custo, robustez e a herança de características úteis tanto do paradigma de redes "ad hoc" como do paradigma de rede sem infraestruturada. Como toda rede sem fio, as redes *mesh* apresentam desafios para os administradores de rede em função da dinâmica e das características peculiares do meio de transmissão [1]. Um destes desafios é aproveitar os recursos de todos os enlaces para fornecer encaminhamento de pacotes, maximizando a vazão da rede. O uso de *Software Defined Networks* (SDN) [14], onde um controlador toma as decisões de roteamento, permite implementar algoritmos de roteamento que considerem a vazão da rede, com um custo computacional superior àqueles que rodam nos processadores dos dispositivos de rede sem fio, normalmente menos potentes e com menos memória.

Neste trabalho implementamos e comparamos algoritmos de roteamento de fluxo não divisível máximo *multicommodity* e custo mínimo. Cada demanda, isto é, cada fluxo entre uma origem e um destino é uma *commodity* de rede. Na seção 1 apresentamos a teoria relacionada ao problema do fluxo não divisível. Restringimos nossa análise a fluxos *unicast*. Obter estas rotas consiste na resolução do *Unsplittable Flow Problem*

(UFP). UFP é um problema NP Difícil, assim em função da necessidade de desempenho da rede, os algoritmos executados no controlador SDN devem ser capazes de obter uma resposta rápida. Para isto recorremos a heurísticas, como aquelas apresentadas na seção 2. Desenvolvemos duas abordagens. Analisando heurísticas construtivas (seção 2.1) utilizadas para obter a solução de UFP na literatura, como $RandomizedRounding$, PROUTE, SPROUTE e ESPROUTE, e desenvolvemos uma heurística baseada no algoritmo de $Dijkstra$ que fornece melhores resultados para as instâncias testadas. Implementamos a heurística de busca proposta por Gamst [8] adaptada para o UFP e propusemos três variações, que denominamos $GAMST_1'$, $GAMST_2'$ e $GAMST_2''$.

Rodamos as heurísticas indicadas na seção 2 em diversas instâncias. A descrição destas instâncias, do experimento e dos seus resultados são apresentados na seção 3. Estes algoritmos são comparados em termos de tempo de execução e da qualidade do resultado obtido. Mostramos que nossa heurística construtiva, apesar de mais lenta que o algoritmo de roteamento baseado em $Dijkstra$, apresenta melhores resultados. Obtemos resultados em geral melhores que os demais algoritmos testados em nosso trabalho. A heurística $GAMST_2''$ apresentou melhores resultados que as heurísticas $GAMST$, $GAMST_1'$ e $GAMST_2'$. Propusemos ao final do trabalho uma heurística $GAMST_3'$, que combina $GAMST_2'$ e $GAMST_2''$ e que avaliaremos em trabalhos futuros. Na última seção concluímos e relacionamos os trabalhos futuros.

1. Fluxo em redes

Os algoritmos clássicos de fluxo máximo buscam o maior uso possível das capacidades disponíveis na rede. Diversos algoritmos existem para este problema, como Ford-Fulkerson, Edmonds-Karp [6], Dinic [7] e Goldberg [9]. Estes algoritmos não identificam as unidades de cada fluxo que são transportadas em uma aresta específica na rede, essencial para o cálculo de rotas. Uma solução de roteamento precisa reconhecer o fluxo individual de cada fonte para cada destino em cada aresta do grafo. Desta forma precisamos obter os fluxos *multicommodity* no grafo e conseguir individualizar cada fluxo. Na seção a seguir mostramos como estes problemas de fluxo podem ser resolvidos.

1.1. Fluxo *multicommodity* não divisível

O problema de fluxo *multicommodity* não divisível (*multicommodity Unsplittable Flow Problem*) para uma rede *mesh* com capacidade finita de vazão pode ser enunciado como: Dado um grafo $G = (V, E)$, onde V é o conjunto dos nós da rede e cada enlace de comunicação é representado por uma aresta $e = (u, v) \in E$ com capacidade de fluxo dado por $c(e) = c(u, v) > 0$. No problema do fluxo *multicommodity*, consideramos k pares fonte-dreno (origem-destino) não necessariamente disjuntos, onde cada par de nós fonte-dreno (s_i, t_i), $\forall i = 1, ..., k$, solicita da rede uma demanda de fluxo de $d_i > 0$ e possui um peso (ou ganho) de w_i. Representamos esta demanda como um conjunto de quadruplas $T = \{(s_1, t_1, d_1, w_1), \ ... \ (s_k, t_k, d_k, w_k)\}$. O objetivo é encontrar o subconjunto de maior peso $T' \subseteq T$, tal que a demanda de cada par pode ser roteada em um único caminho respeitando as restrições de capacidade. Este problema pode ser generalizado considerando que o fluxo (pacotes) pode seguir no máximo l caminhos. Neste trabalho nos restringimos aos fluxos *unicast*, isto é, uma fonte s_i transmite para um único destino t_i utilizando um único caminho. Utilizamos neste artigo a notação $n = |V|$ e $m = |E|$.

Alguns algoritmos consideram uma rede sem gargalos (*no-bottleneck assumption*), isto é, somente consideram demandas onde $d_{max} \leq c_{min}$, onde $d_{max} = \max_i d_i$ e $c_{min} = \min_{e \in E} c(e)$. Outros algoritmos consideram todas as demandas, restringindo, escalando ou escalonando as demandas para permitir seu roteamento. UFP pode estar sujeito a outras variações. Na formulação clássica a demanda d_i é atendida completamente ou não existe o fluxo. Outra formulação considera o atendimento parcial da demanda. Outra variação ocorre quando desejamos uma demanda mínima. Escalonamento, demanda mínima e rede com gargalos serão considerados em trabalhos futuros.

Os problemas de UFP são NP difíceis. O problema da mochila clássico (KNAPSACK) pode ser reduzido para UFP, considerando UFP com uma única aresta $e = (u, v)$ de capacidade $c(e) = W$ igual a capacidade W da mochila. A demanda d_i é igual ao peso p_i para o item i do KNAPSACK e $w_i = 1$. Desta forma, a menos que $P = NP$, não existe uma solução polinomial para o problema do UFP.

1.2. Trabalhos relacionados

Sob a hipótese de uma rede sem gargalos, Kleinberg [11] apresentou um algoritmo aproximado para o UFP, que executa em $O\left(\sqrt{m}\frac{\max c(e)}{\min d_i}\right)$ e que foi melhorado por Baveja and Srinivasan [3] para $O(\sqrt{m})$, usando um algoritmo baseado em programação linear. Raghavan and Tompson [18] resolvem UFP utilizando relaxamento do problema expresso como programação linear. Estes autores mostram que utilizando o recurso de arrendondamento aleatório é possível obter uma aproximação constante se a demanda é limitada a c_{min}/K, quando $K \geq \log n$. Azar and Regev [2] forneceram um algoritmo combinatório mais simples e com a mesma garantia de aproximação. Kolliopoulos and Stein [12] apresentaram a primeira aproximação não trivial para o problema do UFP onde cada solicitação (*commodity*) tem um ganho w_i associado e o objetivo é maximizar o ganho total de pedidos aceitos. Este algoritmo é $O\left(\log m\sqrt{m}\right)$.

Guruswami et al. [10] mostram que em redes dirigidas é NP-difícil aproximar a UFP dentro de um fator de $m^{\frac{1}{2}-\epsilon}$, para qualquer $\epsilon > 0$. Azar and Regev [2] provam que, para as redes sem gargalos, é NP-difícil aproximar uma UFP com ganhos dentro de $\Omega\left(m^{1-\epsilon}\right)$, para qualquer $\epsilon > 0$. Chekuri and Khanna [5] melhoraram um algoritmo guloso para o problema dos caminhos em arestas disjuntas, observando que, em função do número de vértices, o melhor limite inferior é $\Omega(n^{\frac{1}{2}-\epsilon})$ e que o melhor limite superior é $O(n)$. Eles provaram que o algoritmo guloso é $O(n^{\frac{2}{3}})$-aproximado em grafos não direcionados e $O(n^{\frac{4}{5}})$-aproximado em grafos direcionados. Mostraram ainda que existem instâncias em grafos direcionados ou não em que a razão de aproximação do algoritmo guloso é $\Omega(n^{\frac{2}{3}})$. Estes limites se aplicam para o problema UFP de capacidade unitária. Com base neste trabalho, Kolman [13] provou os mesmos limites para um UFP geral com demandas e capacidades arbitrárias. Leighton and Rao [15] melhoraram o limite superior para a relação de aproximação do algoritmo guloso em grafos direcionados para $O(n^{\frac{3}{4}})$. Varadarajan and Venkataraman [19] melhoraram este resultado, obtendo $O\left[(n \times logn)^{\frac{2}{3}}\right]$, quase alcançando o limite inferior $\Omega(n^{\frac{2}{3}})$.

Neste trabalho avaliamos as heurísticas propostas por Raghavan and Tompson [18] e Azar and Regev [2], comparando com uma heurística baseada em menor caminho e com nossa heurística MYHEURISTIC. Avaliamos ainda uma heurística de busca local

proposta por Gamst [8]. Apresentamos três variações para esta heurística na seção 2.2.

2. Algoritmos

Apresentamos na seção 2.1 os algoritmos construtivos e na seção 2.2 os algoritmos de busca local utilizados neste trabalho.

2.1. Heurísticas construtivas

Neste trabalho implementamos e analisamos seis heurísticas construtivas a seguir.

RR: Raghavan and Tompson [18] propõem uma aproximação, denominada *Randomized Rounding*, que faz o relaxamento do problema de programação linear inteira do UFP, utilizando um mecanismo de arredondamento aleatório. RR possui três etapas: (1) obter a solução não inteira para o problema de fluxo usando programação linear; (2) realizar "path stripping"; e (3) selecionar o caminho de forma aleatória.

PROUTE: Azar and Regev [2] apresentam um algoritmo, denominado PROUTE, para o problema do UFP clássico, que divide as demandas em dois conjuntos disjuntos T_1 e T_2, calcula as soluções para cada subconjunto e retorna a melhor das duas. PROUTE utiliza uma função auxiliar denominada ROUTINE2. Esta função utiliza uma variável α para paulatinamente buscar caminhos capazes de atender às demandas, escaladas por α, em cada subconjunto T_1 e T_2. Por sua vez, ROUTINE2 utiliza a função ROUTINE1 que busca um caminho P viável, onde o fluxo j em um caminho P é maior que α, para cada demanda do subconjunto. Um algoritmo de menor caminho é utilizado para obter o caminho P. Uma demanda é aceita se o caminho entre s_i e t_i obtido for menor que um fator $\frac{w_j}{d_j \alpha}$. A cada iteração, as arestas que já tenham atingido sua carga máxima são removidas.

SPROUTE: ROUTINE2 não é fortemente polinomial [2]. SPROUTE utiliza ROUTINE3 (uma variação de ROUTINE2), que limita o custo das arestas e desconsidera as demandas menores que $\frac{w_{max}}{k}$, onde k é a quantidade de requisições. Com isto os autores conseguem limitar as iterações à $O(\log n + \log k)$ mantendo a razão de aproximação de $O(\sqrt{m})$.

ESPROUTE: Para tratar o caso onde as demandas são maiores que a menor capacidade de aresta no grafo, Azar and Regev [2] utilizam uma abordagem semelhante a PROUTE e SPROUTE, contudo dividindo as demandas em um conjunto maior de partições. Este algoritmo é $O\left(\sqrt{m}\log\left(2+\frac{d_{max}}{u_{min}}\right)\right)$-aproximado.

DIJKSTRA: Algoritmos de roteamento interno tradicionais em redes TCP-IP, como OSPF, calculam o menor caminho entre a origem e o destino. Em função disto, implementamos um algoritmo semelhante. Chamamos o algoritmo utilizado de DIJKSTRA. Para obtenção do menor caminho utilizamos o algoritmo de $Dijkstra$, com uma implementação da fila de prioridade baseada em *Heap*. Desta forma o tempo de execução do algoritmo é $O\left(mk + nk\log m\right)$. Consideramos que o fluxo de uma demanda j, roteada por um caminho P_j, atingirá $f_j = min\{d_j, c(e)/num\_fluxos(e)\}$, $\forall e \in P_j$. Com isto supersimplificamos o funcionamento do controle de congestionamento em fluxos TCP em redes reais, pois nosso modelo pode deixar uma aresta subutilizada. Como a demanda pode não ser completamente atendida, o ganho é $p'_j = p_j \times \frac{f_j}{d_j}$ e o benefício total é a soma de todos os p'_j.

MYHEURISTIC: Propusemos uma heurística que utiliza algumas ideias de DIJKSTRA - rotear a demanda pelo menor caminho capaz de suportá-la e considerar o benefício do atendimento parcial desta demanda, pois limitamos a demanda ao valor da aresta de maior capacidade do grafo. MYHEURISTIC roteia primeiro as demandas com maior benefício por demanda, como feito em ROUTINE1. O roteamento considera somente por arestas que não estejam completamente carregadas. Como em DIJKSTRA, o tempo de execução é $O\left(mk + nk \log m\right)$, podendo ser melhorado utilizando "Fibonacci Heap".

2.2. Busca Local

O problema de *Multi-Commodity k-splittable Maximum Flow* (MCkMFP) é um problema NP-difícil que pode ser resolvido usando uma heurística de busca local. A solução de MCkMFP pode ser feita pelo algoritmo proposto por Gamst [8]. A heurística de pesquisa local contém uma rotina interna que busca encontrar iterativamente o caminho mais curto para roteamento das demandas e na atribuição de fluxo ao caminho. Uma solução de MCkMFP onde $k = 1$ é uma solução viável (não ótima) para UFP, pois obtém o fluxo máximo, mas não minimiza o custo.

O algoritmo de GAMST pode ser adaptado para UFP. A alteração que fizemos no algoritmo é utilizar o resultado gerado (os caminhos identificados e o fluxo que foi possível encaminhar) para calcular a função objetivo do problema do fluxo indivisível. Esta versão adaptada do algoritmo de Gamst, que denominamos $GAMST$, realiza a busca aleatória entre as *commodities* e a melhor solução é aquela com maior fluxo. Propusemos duas variações que maximizam o benefício, considerando a fração da demanda atendida. Consideramos duas abordagens: (1) em $GAMST_1{}'$ as demandas são avaliadas de forma aleatória e (2) as demandas são avaliadas em ordem. A ordem é decrescente em função da razão do benefício pela demanda em $GAMST_2{}'$ e crescente em $GAMST_2{}''$.

3. Experimentos

3.1. Implementação

Nós testamos os algoritmos descritos na seção 2 utilizando as instâncias mostradas na Tabela 1, obtidas em *Survivable Network Design Library* (SNDlib1.0) [17][1]. SNDLib é uma biblioteca de instâncias de teste para o projeto de redes de telecomunicações fixas. Cada instância é caracterizada nesta tabela pelo número de nós $n = |V|$, o número de enlaces $m = |E|$ e o número de demandas de tráfego k. As instâncias são fornecidas com um conjunto de demandas e os custos associados aos enlaces. Como as instâncias não fornecem um benefício para as demandas, consideramos nos nossos experimentos que o benefício é 1 (um) para todas as demandas. As instâncias possuem arquivos de configuração que indicam se existem caminhos proibidos, se o grafo criado deve ser direcionado ou não, entre outros. Nos nossos experimentos não consideramos estas restrições e todos os grafos criados foram não direcionados. As instâncias são estáticas, isto é, formam uma infraestrutura de rede onde os nós não apresentam mobilidade. Esperamos estudar em trabalhos futuros os aspectos de mobilidade não considerados no presente trabalho, utilizando outras instâncias ou aplicando a estas modelos de mobilidade da literatura.

Os algoritmos foram implementados em Python versão 2.7 e executados em um computador com Intel(R) Xeon(R) CPU E5-2630 @ 2.30GHz de 24 cores e 128GB de

[1] http://sndlib.zib.de/home.action

Tabela 1. Instâncias SNDlib utilizadas

instância	n	m	k
abilene	12	15	132
atlanta	15	22	210
brain	161	332	14311
cost266	37	57	1332
dfn-bwin	10	45	90
di-yuan	11	42	22
france	25	45	300
geant	22	36	462

instância	n	m	k
germany50	50	88	662
giul39	39	172	1471
india35	35	80	595
janos-us	26	84	650
newyork	16	49	240
nobel-eu	28	41	378
nobel-germany	17	26	121
nobel-us	14	21	91

instancia	n	m	k
norway	27	51	702
pdh	11	34	24
pioro40	40	89	780
polska	12	18	66
sun	27	102	67
ta1	24	51	326
ta2	65	108	1614
zib54	54	80	1246

RAM DDR3. Todas as implementações são sequenciais, portanto cada execução utiliza somente um core do computador. Executamos 20 vezes cada algoritmo em cada instância.

3.2. Heurísticas construtivas

Realizamos testes com o algoritmo *Randomized Rounding* (RR). Rodamos RR para as instâncias mostradas na Tabela 2. Em função do longo tempo de execução, RR foi executado somente 5 vezes para cada instância. Mostramos na Tabela 2 uma comparação entre o tempo de execução médio para RR com o tempo de execução de PROUTE para algumas instâncias. Não apresentamos os intervalos de confiança, pois eles foram pequenos e sem interseção para qualquer valor. Esta diferença de tempo pode ser explicada, porque, apesar do sistema de equações poder ser escrito em tamanho polinomial, o número de variáveis pode ficar muito grande. Por exemplo, apesar da instância abilene possuir 12 vértices, 15 arestas e 132 demandas, o sistema de equações é formado por 2112 variáveis (132 demandas x_i e 1980 fluxos em arestas - $f_{i,e}$). Não efetuamos outras análises da heurística proposta por Raghavan and Tompson [18] em função do alto tempo para obtenção dos resultados.

Tabela 2. Tempo de execução médio de RR e PROUTE - algumas instâncias

Instância	**RR** (em segundos)	**PROUTE** (em segundos)	**Relação (RR/PROUTE)**
abilene	211,8426	0,1480	1431
atlanta	2676,0571	0,1881	14227
dfn-bwin	737,2981	0,1903	3874
di-yuan	17,3467	0,0289	600
pdh	12,0846	0,0204	592

Na Figura 1 apresentamos o ganho total (função objetivo) relativo ao ganho do melhor algoritmo construtivo para cada instância. A escala das ordenadas é logarítmica. Observamos que nossa heurística, na maioria das situações, obtém ganhos melhores que os demais algoritmos. DIJKSTRA não consegue obter os ganhos nas situações onde uma aresta é um gargalo, sendo utilizada por mais de um fluxo. Este problema não acontece com MYHEURISTIC, que busca nesta situação um caminho alternativo, uma vez que o gargalo é removido do grafo pesquisado.

MYHEURISTIC apresenta piores ganhos, por exemplo em $giul39$, quando há desconexão entre a origem s_i e o destino t_i, em função da remoção de uma aresta carregada. O ganho de DIJKSTRA também cai nesta situação, pois uma aresta carregada implica na transferência de menos fluxo. O resultado de DIJKSTRA reportado é pior do que seria em uma rede real, em função da forma como consideramos a divisão dos fluxos pela aresta. Vemos neste caso que PROUTE, SPROUTE e ESPROUTE conseguem resolver

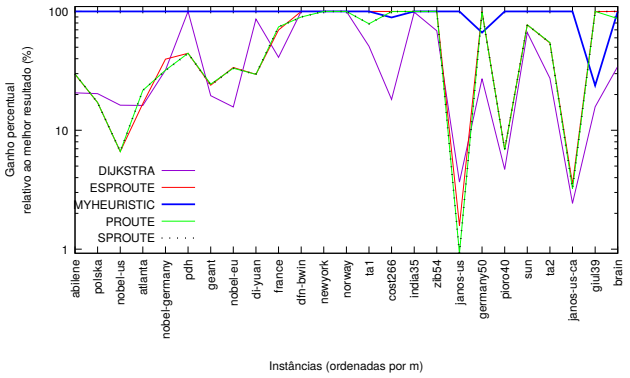

Figura 1. Ganho total para cada instância relativo ao melhor ganho entre os algoritmos

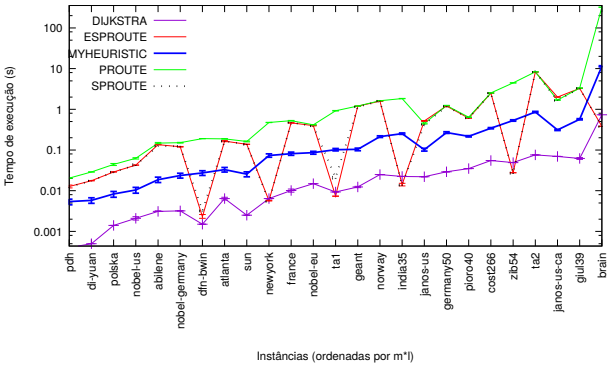

Figura 2. Tempo total de execução entre os algoritmos construtivos para cada instância

o problema obtendo um ganho melhor que DIJKSTRA e MYHEURISTIC. Destacamos ainda que este problema fica mais evidente para DIJKSTRA quando o número de arestas e o número de demandas cresce (instâncias à direita da figura). Nestes casos, DIJKSTRA é consistentemente pior que os demais algoritmos. Em instâncias como por exemplo em $nobel\text{-}us$, os algoritmos PROUTE, SPROUTE e ESPROUTE obtêm um desempenho pior por não conseguirem obter caminhos alternativos para os fluxos. Desta forma demandas são descartadas e assim não conseguimos obter nem uma fração do seu benefício. DIJKSTRA obtém nesta situação um resultado melhor, pois apesar de haver uma redução dos fluxos, em função de consideramos que o benefício é proporcional ao fluxo obtido, o resultado em DIJKSTRA consegue superar PROUTE, SPROUTE e ESPROUTE. Nesta situação, MYHEURISTIC além de obter um benefício fracionário, que incrementa o ganho total, consegue achar outros caminhos menos congestionados que DIJKSTRA.

Na Figura 2 mostramos o desempenho, medido pelo tempo de execução do algoritmo, com o intervalo de confiança de 95% para cada instância. A escala das ordenadas é logarítmica e as instâncias foram ordenadas em função do produto do número de arestas do grafo m e da quantidade de demandas k. Esta ordem foi escolhida para mostrar uma tendência de crescimento no tempo de execução em função destes dois parâmetros. Observamos que os tempos de DIJKSTRA e MYHEURISTIC são semelhantes, porém MYHEURISTIC é mais lento que DIJKSTRA. Este comportamento é esperado uma vez que as ordens de complexidade dos dois algoritmos são iguais, mas as constantes em MYHEURISTIC são maiores por considerar a carga das arestas em cada iteração.

PROUTE é polinomial [2], contudo nos surpreendeu o fato do algoritmo ser consistentemente mais lento que MYHEURISTIC. Analisando nosso código, podemos notar que esta lentidão está relacionada com a manipulação de conjuntos no Python e por fazermos uma cópia completa do grafo a cada execução de ROUTINE1. Notamos ainda que PROUTE segue DIJKSTRA, uma vez que ROUTINE1 utiliza o algoritmo de $Dijkstra$ para achar o menor caminho. SPROUTE e ESPROUTE seguem PROUTE, mas apresentam um desempenho melhor. Este comportamento também é esperado, pois ambos os algoritmos, apesar de utilizarem variações de ROUTINE1 e ROUTINE2 de PROUTE, fazem um tratamento das demandas para redução da quantidade de iterações.

Notamos instâncias onde SPROUTE e ESPROUTE reduzem significativamente

o tempo de execução em relação a PROUTE, por exemplo em $dfn\text{-}bwin$, $newyork$ e $india35$. Nestas instâncias os algoritmos obtêm melhores benefícios com redução de tempo, em função do melhor particionamento das demandas, o que melhora a decisão de qual demanda rotear, diminuindo a quantidade de execuções do algoritmo de menor caminho. Por exemplo em $dfn\text{-}bwin$ o particionamento feito por ESPROUTE, além de remover demandas inviáveis, permite obter ganho equivalente com um tempo menor.

3.3. Busca Local

Em função da natureza aleatória de $GAMST$ e $GAMST_1{}'$, cada experimento consiste em 200 execuções para cada instância e foi utilizado o melhor valor obtido nestas execuções para avaliar o ganho total obtido no roteamento. Mostramos na Figura 3 os resultados para $GAMST$, $GAMST_1{}'$ e $GAMST_2{}'$. Neste gráfico, o eixo das ordenadas é apresentado em escala logarítmica, o que permite destacar as diferenças entre os algoritmos.

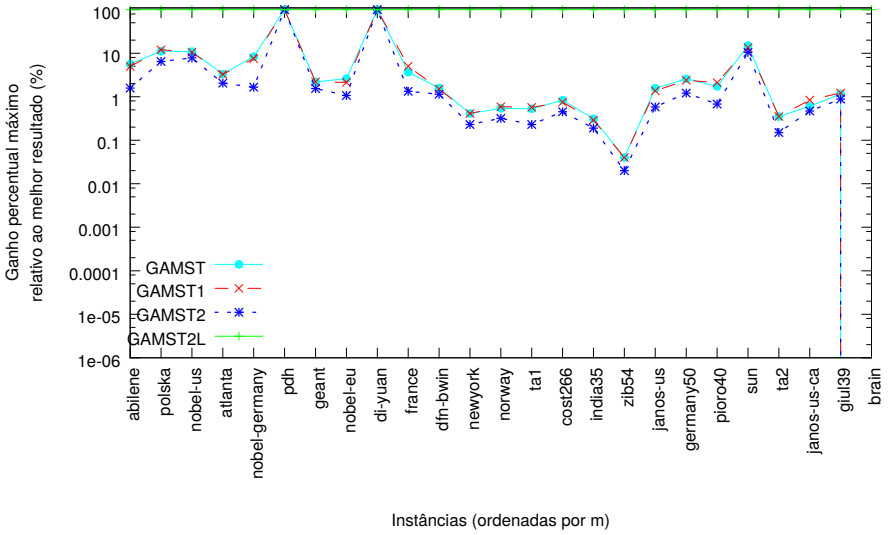

Figura 3. Ganho total em relação ao melhor algoritmo para cada instância em percentual para $GAMST$, $GAMST_1{}'$, $GAMST_2{}'$ e $GAMST_2{}''$

Nas instâncias pdh e $di\text{-}yuan$ os algoritmos $GAMST$, $GAMST_1{}'$ e $GAMST_2{}'$ obtêm o mesmo resultado. Estas são as instâncias mais fáceis do conjunto, pois todas as demandas podem ser roteadas, como pode ser visto na figura 3. Notamos que o valor obtido por $GAMST_1{}'$ é, na maioria das vezes, pior que os outros dois algoritmos. $GAMST_2{}'$ é *quasi-determinístico*, enquanto os outros dois algoritmos exploram aleatoriamente o espaço de solução. $GAMST_2{}'$ não é totalmente determinístico em função da forma que o Python trata o acesso ordenado às chaves de um dicionário. O Python não preserva a ordem das chaves em pesquisas a um dicionário ao gerar a lista ordenada. Por exemplo, para duas demandas i e j com a mesma chave de ordenação $k_i = k_j$, as demandas podem ser colocadas primeiro i e depois j ou o inverso. Em situações onde não existem valores iguais, como por exemplo em $brain$, o resultado é determinístico.

Analisando a execução de $GAMST_2{}'$ e observando que em nossos experimentos o ganho é igual a um para todas as demandas, vemos que a execução de $GAMST_2{}'$ prioriza as *commodities* com maior demanda, assim rapidamente as capacidades das arestas são esgotadas e portanto o algoritmo retorna um resultado pior. Para verificar esta hipótese, alteramos o algoritmo para que a ordenação fosse crescente e chamamos esta variação de $GAMST_2{}''$. A Figura 3 mostra também os valores obtidos por $GAMST_2{}''$, como

GAMST2L. Vemos que $GAMST_2''$ é muito superior aos outros três algoritmos. Em $brain$ o ganho obtido por $GAMST$ é 0,000000248% do valor obtido por $GAMST_2''$. Não só $GAMST_2''$ apresenta melhores valores para o ganho total, como também ele permite que mais demandas possam ser roteadas na rede. A figura 4 mostra a quantidade de demandas atendidas por cada algoritmo para cada instância de teste, com os respectivos intervalos de confiança de 95%. O eixo das ordenadas é logarítmico, pois a quantidade de demandas atendidas por $GAMST_2''$ é, na maioria dos casos, da ordem de 10 vezes maior que o segundo melhor algoritmo. Se consideramos os intervalos de confiança, vemos que $GAMST$ e $GAMST_1'$ apresentam resultados iguais para as instâncias testadas.

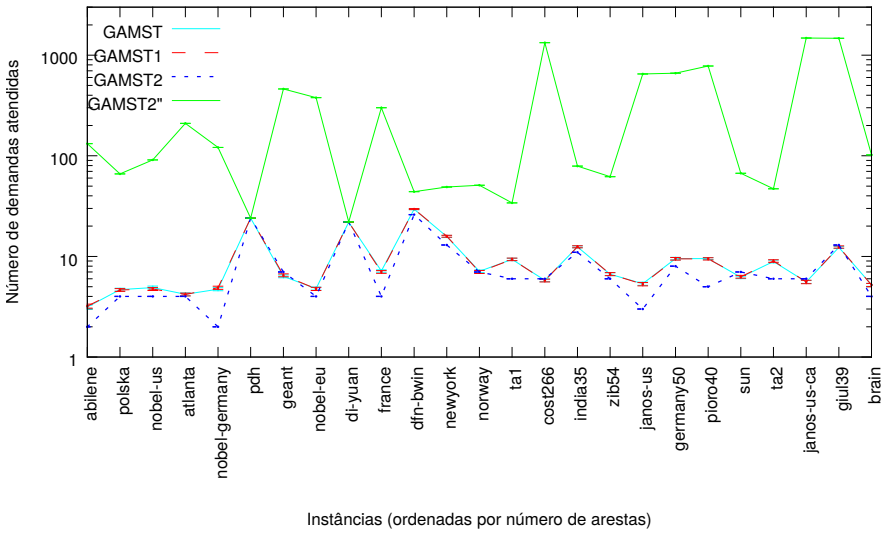

Figura 4. Quantidade de demandas atendidas para cada instância

Na figura 5 apresentamos o tempo médio de execução dos algoritmos para cada instância e seu intervalo de confiança, para 200 execuções para cada instância. Vemos que entre os algoritmos o tempo de execução cresce de $GAMST$ para $GAMST_1'$ e deste para $GAMST_2'$. Este comportamento é esperado, pois acrescentamos complexidade aos algoritmos nesta ordem. $GAMST_1'$ passa a ter uma fila de prioridades que é atualizada em cada iteração no procedimento principal do algoritmo. $GAMST_2'$ além desta fila também executa uma ordenação. Como em ambos os casos as complexidades acrescentadas são polinomiais, as curvas dos três algoritmos são semelhantes. É importante destacar que $GAMST$ e $GAMST_1'$ são aleatórios e assim precisamos rodá-los, no mínimo, 20 vezes para conseguir obter, em média, um valor igual ou superior a $GAMST_2'$. Esta é a ordem de grandeza da diferença de tempo de $GAMST_2'$ para o $GAMST$, assim podemos dizer que $GAMST_2'$ tem um tempo de execução comparável a $GAMST$ obtendo resultados semelhantes.

$GAMST_2''$ possui ordem de complexidade igual a de $GAMST_2'$, afinal somente trocamos as ordenações de decrescentes para crescentes. Contudo a figura 5 apresenta um tempo médio de execução superior para $GAMST_2''$. Este comportamento pode ser explicado mediante uma analise amortizada das execuções. Como $GAMST_2'$ possui, nas instâncias analisadas, muito menos demandas para rotear que $GAMST_2''$. $GAMST_2''$ executa muito mais vezes as mesmas linhas de código, portanto possui uma constante maior. Mesmo nas instâncias onde as demandas possíveis são totalmente atendidas, como em pdh e $di\text{-}yuan$, $GAMST_2''$ executa em tempo superior. Os demais algoritmos esgo-

tam mais rapidamente a capacidade das arestas, até um ponto onde o roteamento na rede é inviável. Assim estes algoritmos finalizam mais rapidamente que $GAMST_2''$.

$GAMST_2''$ obtém para as instâncias de teste resultados muito superiores, pois o ganho das demandas é equiprovável (ganho é igual a 1). Isto não aconteceria se as demandas de grande volume de tráfego também possuíssem alto ganho associado. Isto sugere que podemos criar um algoritmo $GAMST_3'$, que retorna o melhor resultado entre a $GAMST_2'$ e $GAMST_2''$. O tempo de execução de $GAMST_3'$ pode ser considerado como limitado pelo dobro do tempo do algoritmo mais demorado, e ainda polinomial.

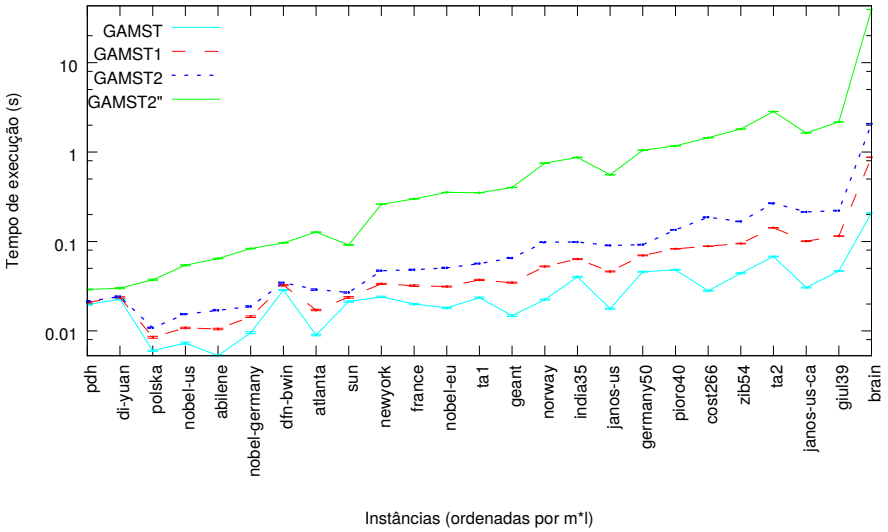

Figura 5. Tempo de execução dos algoritmos de busca para cada instância

Na figura 6 mostramos o percentual de fluxo obtido por cada algoritmo para cada instância em relação ao melhor algoritmo. Os valores são mostrados em escala logarítmica nas ordenadas, para permitir que vejamos melhor as diferenças, que superam 100 vezes em alguns casos. Como era de se esperar, $GAMST_2''$ apresentou os melhores resultados, seguido por $GAMST$, afinal o objetivo deste algoritmo é obter o fluxo máximo em menores caminhos.

4. Conclusões e Trabalhos Futuros

Implementamos e testamos os algoritmos PROUTE, SPROUTE e ESPROUTE propostos por Azar and Regev [2] e os comparamos com dois algoritmos baseados em menor caminho: DIJKSTRA e MYHEURISTIC. DIJKSTRA funciona como os protocolos de roteamento interno tradicionais, enquanto MYHEURISTIC considera a carga das arestas procurando caminhos alternativos quando uma aresta fica sobrecarregada. MYHEURISTIC apresentou na maioria das situações ganhos melhores que os demais algoritmos e um tempo de execução intermediário, nem tão rápido quanto DIJKSTRA, nem tão lento quanto os demais.

Observamos que os algoritmos de fluxo como PROUTE, SPROUTE, ESPROUTE e MYHEURISTIC fornecerem resultados melhores à medida que o número de demandas e arestas cresce no grafo de rede, contudo com um aumento do tempo de execução da ordem de 10 vezes, no pior caso, mas com o benefício do aumento de vazão ao utilizá-los. Estes algoritmos permitem incorporar no cálculo do fluxo as demandas das *commodities*, as capacidades das arestas e as restrições de custo e/ou ganho na função objetivo.

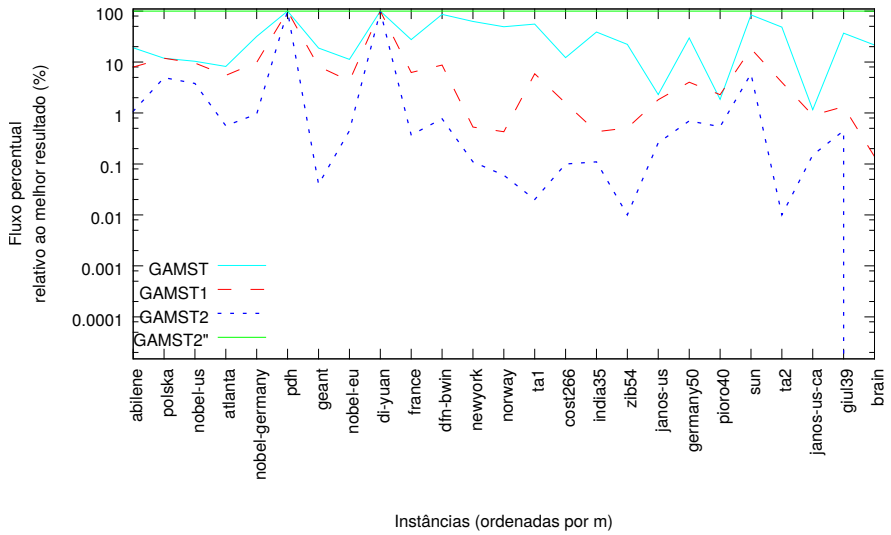

Figura 6. Percentual de fluxos atendidos em relação ao melhor algoritmo de busca para cada instância

Todos os algoritmos construtivos são polinomiais. PROUTE, SPROUTE e ESPROUTE podem ser paralelizados facilmente executando cada partição em um processador diferente. Já DIJKSTRA e MYHEURISTIC não apresentam esta característica. Paralelizar $Dijkstra$ pode beneficiar os demais algoritmos. Em trabalhos futuros exploraremos a paralelização dos algoritmos, pois o controlador SDN possui poder computacional para execução de processos em paralelo.

Implementamos e testamos o algoritmo proposto por Gamst [8] adaptado para UFP e o comparamos com três variações - $GAMST_1{}'$, $GAMST_2{}'$ e $GAMST_2{}''$. Todos estes algoritmos são polinomiais. Para nossas instâncias de teste, $GAMST_2{}''$ apresenta os melhores resultados. $GAMST$ e $GAMST_1{}'$ possuem uma forte componente aleatória, assim podem apresentar melhores resultados que $GAMST_2{}''$, contudo este resultado não foi observado em 200 execuções. Propusemos uma quarta variação, também polinomial, denominada $GAMST_3{}'$. Esperamos em trabalhos futuros avaliá-la, a fim de verificar se a qualidade dos resultados é superior aos demais algoritmos utilizados neste trabalho.

Em um ambiente de rede sem fio em malha, os roteadores são normalmente fixos, enquanto as estações (as demandas) são móveis. A dinamicidade das estações não foi considerada neste trabalho. Na literatura, existem algoritmos que resolvem UFP *online*, isto é, à medida que novas demandas são acrescentadas ou retiradas. Esta abordagem deverá ser analisada em trabalhos adicionais. O presente trabalho executou os algoritmos em um ambiente simulado. Pretendemos avaliá-los em um ambiente real implementando-os sobre a plataforma Ethanol [16].

Agradecimentos

Agradecemos às instituições de amparo à pesquisa CNPq, CAPES e FAPEMIG pelo financiamento e suporte para o desenvolvimento dessa pesquisa.

Referências

[1] Akyildiz, I. F., Wang, X., and Wang, W. (2005). Wireless mesh networks: A survey. *Comput. Netw. ISDN Syst.*, 47(4):445–487.

[2] Azar, Y. and Regev, O. (2001). Strongly polynomial algorithms for the unsplittable flow problem. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 15–29. Springer.

[3] Baveja, A. and Srinivasan, A. (2000). Approximation algorithms for disjoint paths and related routing and packing problems. *Mathematics of Operations Research*, 25(2):255–280.

[4] Bruno, R., Conti, M., and Gregori, E. (2005). Mesh networks: Commodity multihop ad hoc networks. *IEEE Communications Magazine*, 43(3):123–131.

[5] Chekuri, C. and Khanna, S. (2003). Edge disjoint paths revisited. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 628–637. Society for Industrial and Applied Mathematics.

[6] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2009). *Introduction to Algorithms*. The MIT Press, 3nd edition.

[7] Dinitz, Y. (2006). Dinitz'algorithm: The original version and even's version. In *Theoretical Computer Science*, pages 218–240. Springer.

[8] Gamst, M. (2014). A local search heuristic for the multi-commodity k-splittable maximum flow problem. *Optimization Letters*, 8(3):919–937.

[9] Goldberg, A. V. and Tarjan, R. E. (1988). A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940.

[10] Guruswami, V., Khanna, S., Rajaraman, R., Shepherd, B., and Yannakakis, M. (2003). Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. *Journal of Computer and System Sciences*, 67(3):473–496.

[11] Kleinberg, J. M. (1996). *Approximation algorithms for disjoint paths problems*. PhD thesis, Citeseer.

[12] Kolliopoulos, S. G. and Stein, C. (1997). Improved approximation algorithms for unsplittable flow problems. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 426–436. IEEE.

[13] Kolman, P. (2003). A note on the greedy algorithm for the unsplittable flow problem. *Information Processing Letters*, 88(3):101–105.

[14] Kreutz, D., Ramos, F. M. V., Veríssimo, P., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63.

[15] Leighton, T. and Rao, S. (1999). Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832.

[16] Moura, H., Bessa, G. V., Vieira, M. A., and Macedo, D. F. (2015). Ethanol: Software defined networking for 802.11 wireless networks. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 388–396. IEEE.

[17] Orlowski, S., Wessaly, R., Pioro, M., and Tomaszewski, A. (2010). Sndlib 1.0 - survivable network design library. *Netw.*, 55(3):276–286.

[18] Raghavan, P. and Tompson, C. D. (1987). Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374.

[19] Varadarajan, K. and Venkataraman, G. (2004). Graph decomposition and a greedy algorithm for edge-disjoint paths. In *In Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA*, pages 379–380.