UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS DE SOUSA DE OLIVEIRA

ALGORITMO GENÉTICO PARA O ROTEAMENTO DE TRÁFEGOS ELEFANTE EM REDES DE DATA CENTER

QUIXADÁ

2022

LUCAS DE SOUSA DE OLIVEIRA

ALGORITMO GENÉTICO PARA O ROTEAMENTO DE TRÁFEGOS ELEFANTE EM REDES DE DATA CENTER

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Críston Pereira de Souza

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

O48a Oliveira, Lucas de Sousa de.

Algoritmo genético para o roteamento de tráfegos elefante em redes de data center / Lucas de Sousa de Oliveira. – 2022.

46 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2022.

Orientação: Prof. Dr. Críston Pereira de Souza.

1. Roteadores (Redes de computadores). 2. Algoritmo genético. 3. Otimização. I. Título.

CDD 004

LUCAS DE SOUSA DE OLIVEIRA

ALGORITMO GENÉTICO PARA O ROTEAMENTO DE TRÁFEGOS ELEFANTE EM REDES DE DATA CENTER

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: ____/____/________.

BANCA EXAMINADORA

Prof. Dr. Críston Pereira de Souza (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Wladimir Araujo Tavares
Universidade Federal do Ceará (UFC)

Prof. Dr. Jeandro de Mesquita Bezerra
Universidade Federal do Ceará (UFC)

À minha família por todo o apoio que eles me deram.

AGRADECIMENTOS

Agradeço aos meus pais por me apoiarem sempre.

Agradeço ao Professor Críston Pereira de Souza pela paciência e orientação.

Agradeço aos professores Wladimir Araujo Tavares e Jeandro de Mesquita Bezerra pela disponibilidade e por participaram da banca avaliadora.

Agradeço aos amigos que fiz e pessoas que conheci durante esses anos de curso. Em especial: Alessandro, Carlos, Claro, Diego, Diogo, Gabriel, Ravache, Severo e Xavier.

Agradeço a comunidade acadêmica da UFC de Quixadá pelo ensino de altíssima qualidade.

“I wish there was a way to know you’re in the
good old days before you’ve actually left them.”
(Andy Bernard)

RESUMO

Conforme redes de computadores aumentam em tamanho, surgem novos desafios quanto ao roteamento de dados. Se torna importante empregar estratégias que busquem reduzir congestionamento de dados e perda de pacotes. Para oferecer uma alternativa aos algoritmos de roteamento existentes, este trabalho propõe um algoritmo genético para rotear tráfegos elefante em redes de data center. Esse algoritmo busca minimizar a utilização dos *links* em redes com topologia *fat-tree* na tentativa de realizar um balanceamento de carga. Ele foi implementando em um controlador para redes de arquitetura SDN com redes virtuais emuladas. O algoritmo foi testado com diferentes parâmetros para um conjunto de padrões de tráfego. Os melhores resultados, segundo as métricas escolhidas, foram comparados a outros algoritmos de roteamento relacionados. As comparações mostraram que o algoritmo proposto conseguiu melhorar a eficiência das redes testadas para certos padrões de tráfego.

Palavras-chave: Roteadores. Redes de Computadores. Algoritmo Genético. Otimização.

ABSTRACT

As computer networks increase in size, new challenges in routing data arise. It becomes important to employ strategies that seek to reduce data congestion and packet loss. To provide an alternative to existing routing algorithms, this paper proposes a genetic algorithm to route elephant flows in data center networks. This algorithm seeks to minimize the utilization of links in networks with fat-tree topology in an attempt to perform load balancing. It was implemented in a controller for SDN architecture networks with emulated virtual networks. The algorithm was tested with different parameters for a set of traffic patterns. The best results, according to the chosen metrics, were compared to other related routing algorithms. The comparisons showed that the proposed algorithm was able to improve the efficiency of the tested networks for certain traffic patterns.
Keywords: Routers. Computer Networks. Genetic Algorithm. Optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Fat-Tree de 3 camadas 19
Figura 2 – Construção de uma Fat Tree 20
Figura 3 – Fluxo do algoritmo proposto 25
Figura 4 – Cromossomo de exemplo 27
Figura 5 – Caminhos 1001 e 1003 27
Figura 6 – Caminhos 2001 e 2002 28
Figura 7 – Operação de crossover 30
Figura 8 – Taxa de Transferência em Mbps - Fat-Tree com *4* pods 39
Figura 9 – Taxa de Transferência Normalizada - Fat-Tree com *4* pods 40
Figura 10 – Taxa de Perda de Pacotes - Fat-Tree com 4 *pods* 40
Figura 11 – Latência Bidirecional 40
Figura 12 – Taxa de Transferência em Mbps 41
Figura 13 – Taxa de Transferência Normalizada 42
Figura 14 – Taxa de Perda de Pacotes 42
Figura 15 – Latência Bidirecional 42

LISTA DE TABELAS

Tabela 1 – Configuração da máquina de testes 33
Tabela 2 – Padrões de tráfego usados 33
Tabela 3 – Métricas utilizadas no experimento 33
Tabela 4 – Resultados do genético para *fat-tree* com 4 *pods* e tráfego stag_0.2_0.3 . . . 34
Tabela 5 – Resultados do genético para *fat-tree* e com 4 *pods* e tráfego stag_0.4_0.3 . . 35
Tabela 6 – Resultados do genético para *fat-trees* com 4 *pods* e tráfego stag_0.5_0.3 . . 35
Tabela 7 – Resultados do genético para *fat-trees* com 4 *pods* e tráfego random1 35
Tabela 8 – Resultados do genético para *fat-trees* com 4 *pods* e tráfego random2 36
Tabela 9 – Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.2_0.3 . . 36
Tabela 10 – Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.4_0.3 . . 37
Tabela 11 – Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.5_0.3 . . 37
Tabela 12 – Resultados do genético para *fat-trees* com 8 *pods* e tráfego random1 38
Tabela 13 – Resultados do genético para *fat-trees* com 8 *pods* e tráfego random2 38
Tabela 14 – Parâmetros do algoritmo genético para *fat-trees* com 4 *pods* 39
Tabela 15 – Parâmetros do algoritmo genético para *fat-trees* com 8 *pods* 41

LISTA DE ABREVIATURAS E SIGLAS

ECMP	Equal-cost multi-path routing
FCT	Flow Completition Time
GA-ACO	Genetic Algorithm and Ant COlony algorithms
SDN	Software Define Network
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO . . . 13
1.1 Objetivo . . . 14
1.1.1 Objetivo Geral . . . 14
1.1.2 Objetivos Específicos . . . 14
2 TRABALHOS RELACIONADOS . . . 15
2.1 Hedera: dynamic flow scheduling for data center networks . . . 15
2.2 Fincher: Elephant flow scheduling based on stable matching in data center networks . . . 16
2.3 An optimal and dynamic elephant flow scheduling for SDN-based data center networks . . . 17
2.4 Trabalho Proposto . . . 18
3 FUNDAMENTAÇÃO TEÓRICA . . . 19
3.1 Fat-Trees . . . 19
3.2 Tráfegos Elefante e Rato . . . 20
3.3 SDN . . . 21
3.4 Métricas para algoritmos de roteamento . . . 22
3.4.1 Utilização de link . . . 22
3.4.2 Taxa de perda de pacotes . . . 22
3.4.3 Latência Bidirecional . . . 22
3.4.4 Taxa de Transferência . . . 22
3.4.5 Taxa de Transferência Normalizada . . . 23
3.5 Algoritmos Genéticos . . . 23
3.5.1 Seleção . . . 24
3.5.2 Crossover . . . 24
3.5.3 Mutação . . . 24
4 PROCEDIMENTOS METODOLÓGICOS . . . 25
4.1 Algoritmo proposto . . . 25
4.1.1 Visão geral . . . 25
4.1.2 Algoritmo genético . . . 26
4.1.2.1 Codificação . . . 26

4.1.2.2 Geração da população *28*
4.1.2.3 Função de avaliação *28*
4.1.2.4 Operação de seleção *28*
4.1.2.5 Operação de mutação *29*
4.1.2.6 Operação de crossover *29*
4.2 Algoritmo guloso 29
4.3 Implementação dos algoritmos utilizados 32
4.4 Ambiente de testes 33
4.5 Métricas para avaliação 33
5 RESULTADOS 34
5.1 Resultados do algoritmo genético para *fat-trees* com 4 pods 34
5.2 Resultados do algoritmo genético para *fat-trees* com 8 *pods* 36
5.3 Comparação com os algoritmos escolhidos para *fat-trees* com 4 *pods* . . 38
5.4 Comparação com os algoritmos escolhidos para *fat-trees* com 8 pods . . 41
6 CONCLUSÕES E TRABALHOS FUTUROS 43
REFERÊNCIAS 44

1 INTRODUÇÃO

Conforme redes de computadores aumentam em número de máquinas e crescem em complexidade, novos desafios surgem quanto a transmissão de dados em rede. Um desses desafios é o de rotear os dados em rede de maneira eficaz e eficiente. Como garantir que os dados sejam entregues nos destinos corretos, considerando o tamanho de cada demanda e as rotas disponíveis, de modo a reduzir congestionamentos.

Algoritmos de roteamento podem melhorar o desempenho de uma rede ao reduzir gargalos e aumentar a eficiência na entrega de dados. São soluções que podem beneficiar os mais diversos negócios e indivíduos, principalmente considerando a quantidade de redes públicas e privadas ao redor do mundo.

Este trabalho apresentará um algoritmo genético voltado para o roteamento de tráfegos elefante em *fat-trees*. O algoritmo será executado em um controlador remoto para redes de arquitetura SDN (Software Defined Network). A topologia utilizada será implementada usando um emulador de redes virtuais.

A topologia alvo deste trabalho é a *fat-tree*. Essa topologia se baseia em conectar *switches* que variam em número de portas, divididos em níveis diferentes, organizados como uma árvore com múltiplas raízes (LEISERSON, 1985). É uma topologia comum em data centers, por ser barata e fácil de expandir. O número de *hosts* pode ser aumentado ao se adicionar novos *switches* na rede.

Tráfegos elefante são demandas que carregam muita informação (MORI *et al.*, 2004). Quando eles ocorrem com frequência e não são roteadas adequadamente, uma rede pode sofrer gargalos e a entrega de pacotes pode ser comprometida. Por isso é importante identificar esses tráfegos durante o processo de roteamento e tentar roteá-los o melhor possível. Assim, algoritmos com foco em tratar essas categorias de tráfego podem reduzir congestionamentos e melhorar o tráfego em rede.

O algoritmo proposto será comparado a outras técnicas mais conhecidas como ECMP (ZHANG *et al.*, 2015) e Hedera (AL-FARES *et al.*, 2010). Além disso, ele será comparado a um algoritmo guloso arbitrário que emprega uma estratégia simples de rotear os dados pelo caminho que tem o *link* com a maior capacidade disponível entre um conjunto de caminhos escolhidos. Todos os algoritmos utilizados serão testados no mesmo ambiente e comparados segundo as métricas escolhidas.

Nas seções seguintes serão apresentados os objetivos do algoritmo proposto, traba-

lhos relacionados, alguns conceitos importantes para a construção da solução, a metodologia empregada e os resultados alcançados.

1.1 Objetivo

1.1.1 Objetivo Geral

O objetivo deste trabalho é implementar e avaliar um algoritmo genético para o roteamento de tráfegos elefante em *fat-trees*.

1.1.2 Objetivos Específicos

- Implementar um algoritmo guloso que escolhe o caminho com a maior capacidade disponível entre um conjunto de caminhos possíveis para cada demanda
- Testar o algoritmo proposto para diferentes combinações de parâmetros
- Comparar o desempenho da solução (balanceamento de carga) do algoritmo proposto aos algoritmos escolhidos (ECMP, Hedera e Guloso)

2 TRABALHOS RELACIONADOS

Nesta seção serão apresentados alguns trabalhos que abordam conceitos relacionados ao trabalho proposto. No fim, há uma breve comparação entre os trabalhos apresentado e este.

2.1 Hedera: dynamic flow scheduling for data center networks

No trabalho de Al-Fares *et al.* (2010) é apresentado um sistema para roteamento de dados em redes com encaminhamento de dados por múltiplos caminhos e com arquitetura em árvore, chamado Hedera. Ele identifica que os algoritmos existentes, no contexto do trabalho, ignoram o tamanho dos tráfegos ao realizar seus roteamentos, assim gerando mais congestionamentos em rede.

O trabalho discute sobre redes com encaminhamento de dados por múltiplos caminhos e como algoritmos existentes tentam se beneficiar disso. Ele mostra como uma dos métodos comumente empregados, o Equal-cost multi-path routing (ECMP), gera atrasos na rede ao agendar tráfegos de forma arbitrária. Isso acontece porque quando um tráfego é atribuído a uma rota, parte da capacidade do *link* associado é ocupada. Quando não há mais espaço em um *link*, sua rota fica bloqueada para novos tráfegos. Dessa forma, congestionamentos tendem a surgir conforme mais *links* ficam ocupados. Como o ECMP roteia os dados de forma arbitrária, ele não aplica nenhuma estratégia adicional para tentar reduzir congestionamentos.

O objetivo de Al-Fares *et al.* (2010) é apresentar um sistema de roteamento de tráfegos que considera a quantidade de dados transmitida por cada fluxo. É uma solução de agendamento dinâmico que oferece uma alternativa aos algoritmos de abordagem estática que ele cita, como o ECMP.

No trabalho é feita uma análise dos métodos de roteamento existentes e ressalta a natureza estática da tomada de decisão deles. O principal discutido é o ECMP. Um algoritmo de roteamento que atribui pesos iguais a todos os caminhos disponíveis de um tráfego, e escolhe um deles arbitrariamente. Dessa forma, ele tenta se beneficiar das redes em que existem múltiplas rotas, e tenta garantir que mais de uma rota possa ser escolhida. Apesar disso, devido a colisões na escolha dos caminhos, quanto mais tráfegos, maiores as chances de ocorrerem atrasos na rede.

No trabalho é descrita a arquitetura do sistema proposto. O mecanismo de agendamento é composto de 3 etapas. Inicialmente, fluxos grandes são detectados nos *switches* de borda. Depois, os melhores caminhos são calculados para esses fluxos, que por fim são reencaminhados

pelas novas rotas.

O sistema foi testado em um simulador de redes virtuais que modela o comportamento de fluxos TCP. O modelo utilizado foi composto de 8.192 *hosts* em uma arquitetura de *fat-tree*.

No final do trabalho, ele compara o sistema com o ECMP e um *switch* hipotético não bloqueante. A métrica utilizada é a Largura de Bisseção de Banda. Os resultados mostram que para o modelo utilizado nos testes, a solução proposta é 96% ótima e 113% melhor que os métodos que empregam técnicas estáticas de balanceamento.

2.2 Fincher: Elephant flow scheduling based on stable matching in data center networks

No trabalho de Zhang *et al.* (2015) é apresentado o algoritmo Fincher, uma solução para roteamento de tráfegos elefante em *fat-tree*. Semelhante ao trabalho de Al-Fares *et al.* (2010), o algoritmo Fincher se propõe a oferecer um método de agendamento dinâmico como alternativa a algoritmos de balanceamento estáticos como o ECMP. No entanto, ele tenta resolver o problema do Hedera de tráfego desbalanceado e congestionamentos, que pode acontecer por considerar apenas limitações de largura de banda.

Seu objetivo é apresentar um algoritmo capaz de reduzir a latência e evitar congestionamentos em redes de computadores, especificamente redes com topologias *fat-tree*.

Ele utiliza Stable Matching Theory para modelar e resolver o problema do escalonamento de tráfegos. Esse método se baseia na solução do problema do Stable Marriage Problem onde o objetivo é encontrar o emparelhamento estável entre homens e mulheres considerando uma lista de preferências para cada indivíduo (IWAMA; MIYAZAKI, 2008). Nesse problema, a solução é estável quando os pares formados não possuem incentivos para se separar, portanto, quando as preferências foram satisfeitas. O Fincher utiliza a mesma ideia e propõe um método que tenta encontrar um emparelhamento ótimo entre fluxos e *switches*.

Através de uma visão global da rede, o Fincher cria listas de preferências de *switches* e fluxos. Então, ele aplica uma versão modificada do algoritmo Gale-Shapley, o método usado no problema Stable Marriage (IWAMA; MIYAZAKI, 2008).

Para execução e testes, o algoritmo foi implementado na plataforma de software de rede Mininet (MININET, 2018). A simulação dos tráfegos levou em consideração a presença de tráfegos elefante.

Como métrica de comparação, ele utilizou a Flow Completition Time (FCT) que

calcula a duração de cada tráfego. Além disso, ele aplica essa métrica em simulações com o ECMP e Hedera para fins de comparação.

Os resultados mostram uma redução de 14% até 29% do FCT em relação ao ECMP e de 10% até 28% quando comparado com o Hedera. No trabalho é discutido como a redução de congestionamento e de colisões no roteamento produzem resultados melhores quando em relação aos outros algoritmos testados.

2.3 An optimal and dynamic elephant flow scheduling for SDN-based data center networks

O trabalho de Honghui *et al.* (2020) apresenta um método que combina um algoritmo de colônia de formigas com um algoritmo genético para realizar o roteamento de tráfegos elefante em rede, o Genetic Algorithm and Ant COlony algorithms (GA-ACO). Ele se propõe a ser uma alternativa a métodos amplamente adotados como o ECMP, e utiliza a arquitetura de Software Define Network (SDN) como o Hedera.

O objetivo dele é aplicar um algoritmo para roteamento de tráfegos elefante em *fat-trees* para tentar reduzir o congestionamento em uma rede e facilitar seu balanceamento de carga. Além disso, ele utiliza uma abordagem que se aproveita da arquitetura de SDN para obter uma visão global do estado da rede.

Ele difere do ECMP por se tratar de um roteamento dinâmico que também considera o tamanho dos fluxos. Ademais, ele incorpora o tratamento de tráfegos elefante de forma que os tráfegos menores sejam menos afetados, algo que pode ocorrer em algoritmos como o Fincher. Ele também tenta corrigir os problemas de otimização local no algoritmo de colônia de formigas do método ACO-SDN proposto por (RAOUF; ASKR, 2019).

O funcionamento do algoritmo se baseia primeiramente em coletar informação sobre as rotas presentes na rede a todo momento. Para isso, ele utiliza o Sflow (HONGHUI *et al.*, 2020) para monitorar o estado da rede. Quando um fluxo de dados chega na rede, ele utiliza o ECMP para rotear o tráfego. Se a rota escolhida tiver uma utilização superior a um limite especificado, então ele aplica o algoritmo GA-ACO.

O GA-ACO usa um algoritmo genético para escolher caminhos candidatos de acordo com o valor de utilização da rota escolhida pelo ECMP. Depois, ele aplica um algoritmo de colônia de formigas para escolher entre os candidatos o caminho ótimo.

O trabalho apresenta uma comparação do GA-ACO com o ECMP e o SDN. Em relação às métricas utilizadas, os resultados dos testes mostraram que o GA-ACO apresenta um

desempenho superior aos outros algoritmos testados, ECMP e SDN.

2.4 Trabalho Proposto

O trabalho proposto, semelhante aos algoritmos apresentados, busca mostrar uma solução de roteamento dinâmico para tráfegos elefante. Ele também se empenha em se oferecer como uma alternativa ao ECMP. Além disso, ele compartilha da mesma topologia alvo dos algoritmos citados, a *fat-tree*.

O algoritmo proposto consiste em um algoritmo genético cujo objetivo é minimizar a utilização dos *links* de uma rede. Ele é chamado sempre que um tráfego elefante é identificado. Após isso, ele recebe um conjunto de demandas e tenta encontrar a melhor rota para cada demanda considerando o objetivo especificado. Ele busca reduzir congestionamentos ao balancear a carga dos *links* presentes na rede.

Ao contrário do algoritmo ECMP, o algoritmo proposto identifica e roteia tráfegos elefante na tentativa de minimizar possíveis congestionamentos na rede.

Diferente do algoritmo GA-ACO que trabalha com demandas individuais. O algoritmo proposto leva em consideração múltiplas demandas e como o roteamento delas afeta a rede como um todo.

O trabalho proposto será comparado ao Hedera, ao ECMP e a um algoritmo guloso arbitrário.

3 FUNDAMENTAÇÃO TEÓRICA

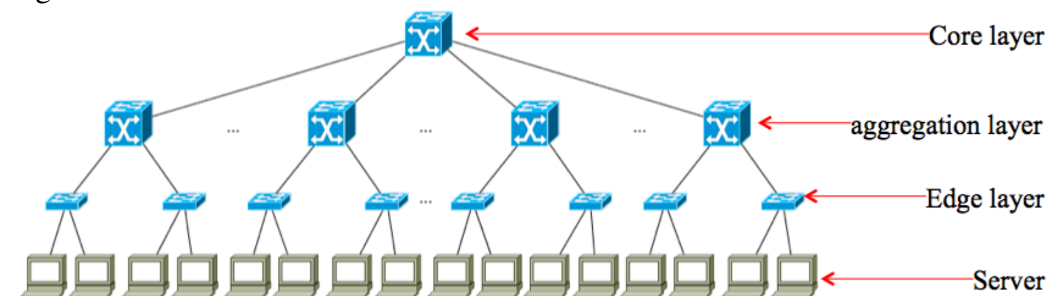
Nesta seção, serão apresentados alguns dos principais conceitos relacionados a este trabalho. São assuntos que serão a base para o desenvolvimento do algoritmo proposto.

3.1 Fat-Trees

Fat-Trees são uma topologia de rede baseada em árvores com múltiplas raízes (HONGHUI *et al.*, 2020). Comumente adotadas em grande data centers, pois podem ser expandidas com facilidade e suportam transmissão de dados por múltiplas caminhos (ZAHAVI *et al.*, 2014).

Essa topologia foi primeiro apresentado por Leiserson (1985). Seu objetivo é conectar *switches* e combinar eles de forma inteligente para interconectar máquinas em rede. Dessa forma é possível expandir a capacidade de uma rede ao adicionar novos *switches*. Ademais, a capacidade de máquinas na rede depende da quantidade de *switches* nas camadas superiores.

Figura 1 – Fat-Tree de 3 camadas



Fonte: Dehury ().

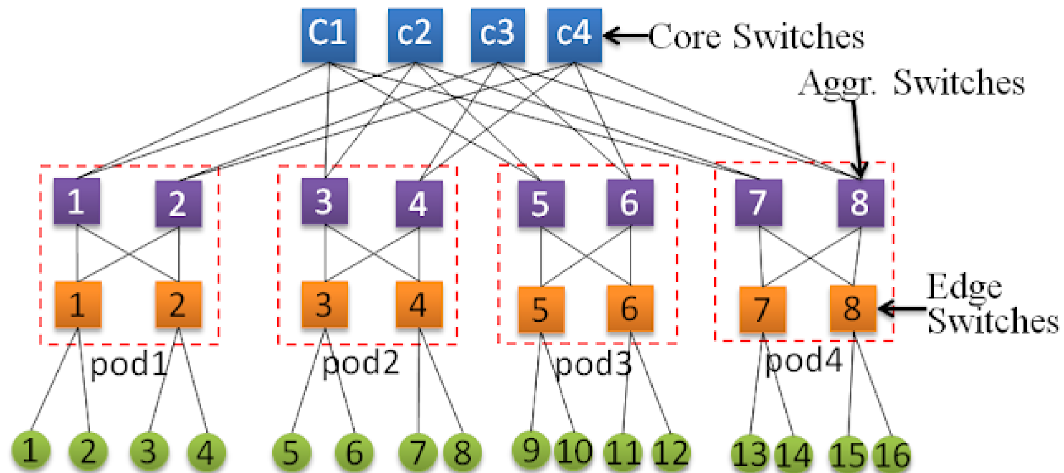
Como explica Alqahtani e Hamdaoui (2018), em *fat-trees* os nós folhas são as máquinas *hosts* da rede, responsáveis por gerar o tráfego. Os demais nós na árvore são ocupados por três tipos distintos de *switches*. No topo e em menor quantidade, ficam os *core switches*. Ele possuem um número de portas reduzido e se conectam a camada de baixo formada pelos *switches* de agregação. Estes por sua vez se conectam aos *switches* de borda, que por fim se conectam as máquinas *hosts*.

Fat-Trees podem ser classificadas pelo número de portas que cada *switch*, em uma rede com essa topologia, possui (ZHANG *et al.*, 2015). Portanto, uma *k-pod* Fat-Tree significa que os *switches* em uma rede com essa topologia possuem *k* portas cada.

Podemos usar o valor *k* de uma *fat-tree* para calcular a quantidade máxima de

switches possíveis em cada nível da árvore. Assim, o maior número possível de *switches core* em uma *k-pod Fat-Tree* é $(\frac{k}{2})^2$. Para *switches* de agregação e de borda a quantidade máxima é $\frac{k^2}{2}$. Além disso, o número máximo de *hosts* possíveis é $\frac{k^3}{4}$.

Figura 2 – Construção de uma Fat Tree



Fonte: Dehury ().

Uma das vantagens desta topologia é que redes que aplicam essa estrutura podem ser escaladas com facilidade sem degradação de desempenho (ABHIJIT; ANWAR, 2019). Isso permite seu uso em data centers onde pode existir a necessidade de se conectar várias máquinas, mantendo requisitos de escalabilidade.

Para garantir a eficiência desta topologia é desejável se utilizar de um algoritmo de roteamento que consiga minimizar congestionamentos e se aproveite da disponibilidade de múltiplas rotas (ZHANG *et al.*, 2015). Caso contrário, tráfegos elefante podem impactar negativamente o desempenho de uma rede.

3.2 Tráfegos Elefante e Rato

Como Chhabra e Kiran (2017) explicam, em uma rede, informações transmitidas podem apresentar tamanhos diferentes. A quantidade de dados sendo enviada de um ponto a outro na rede pode ser utilizada para classificar os tráfegos. Entre as denominações existentes, duas são consistentes no meio acadêmico: Tráfegos Elefante e Rato.

Tráfegos Elefante são demandas que geram cerca de 80% do tráfego total em uma rede (WANG *et al.*, 2014). Demandas são consideradas tráfegos elefante quando seus tamanhos correspondem a mais de 10% da largura de banda de um *link* na rede (HONGHUI *et al.*, 2020).

Tráfegos rato, por outro lado, consistem em fluxos de dados que carregam pouca informação (WANG *et al.*, 2014). Apesar de carregarem menos dados, eles são mais frequentes.

A principal diferença entre esses dois tipos de tráfegos é o impacto que eles podem gerar em uma rede quando não tratados corretamente durante o roteamento. Tráfegos elefante, por exemplo, podem congestionar a rede, pois tendem a ter um tempo de conclusão de fluxo mais longo e ocupam mais da capacidade disponível de um *link*.

Uma forma típica de identificar esses tráfegos é coletar os pacotes que circulam por uma rede para então obter informações estatísticas sobre eles (MORI *et al.*, 2004). Esses dados são analisados e é possível determinar os tráfegos. No entanto, problemas de escalabilidade podem surgir ao se empregar métodos do gênero em *links* de alta velocidade.

Quando identificados é possível dar prioridade aos tráfegos rato permitindo reduzir os congestionamentos em rede (MORI *et al.*, 2004). Para tanto é importante definir uma política de priorização dinâmica onde fluxos menores possam ser identificados de forma automática e recebam preferência em relação aos fluxos maiores. Quando bem aplicado, melhoras podem ser notadas em uma rede de forma global.

3.3 SDN

Uma Software Defined Network (SDN) é uma arquitetura que busca centralizar o controle sobre os fluxos de dados em uma rede (EISSA *et al.*, 2019). É a separação física entre os dispositivos que controlam o tráfego e os dispositivos que encaminham os dados.

A separação entre o plano que controla os dados e o plano por onde os dados trafegam é feita através de uma Application Programming Interface (API) (EISSA *et al.*, 2019). Um exemplo de API do gênero é o OpenFlow.

Switches que oferecem suporte a uma API, como o Open Flow, podem ser acessados por um controlador que é capaz de alterar as configurações do *switch* (EISSA *et al.*, 2019).

Controladores podem ser definidos por software usando aplicações como o *framework* Ryu (RYU, 2018). Nele é possível criar aplicações que interagem com redes que utilizam a arquitetura SDN. Dessa forma é possível implementar soluções de monitoramento ou roteamento de dados que rodam em um controlador remoto.

3.4 Métricas para algoritmos de roteamento

Para se avaliar a qualidade de um algoritmo de roteamento em relação a outros, costuma-se aplicar diferentes métricas.

As seguintes métricas fazem parte do escopo deste trabalho:

– Utilização de *link*
– Taxa de perda de pacotes
– Latência bidirecional
– Taxa de transferência
– Taxa de transferência normalizada

3.4.1 Utilização de link

A métrica utilização de link consiste na razão entre a capacidade disponível em um *link* pela sua capacidade total (Hassidim *et al.*, 2013).

3.4.2 Taxa de perda de pacotes

A taxa de perda de pacotes mede a razão entre a quantidade de pacotes entregues pelo número total de transmitidos em rede durante um certo intervalo de tempo (LEE *et al.*, 2015). Essa métrica permite avaliar a qualidade de uma rede em relação à consistência na transmissão de informação.

No contexto de algoritmos de roteamento, essa métrica permite avaliar o quão bem a estratégia de roteamento escolhida consegue minimizar a perda de pacotes.

3.4.3 Latência Bidirecional

A latência bidirecional apresenta o tempo que um pacote leva para ser entregue a um destino mais o tempo necessário para confirmar sua entrega (HAGIWARA *et al.*, 2001).

3.4.4 Taxa de Transferência

Mede a quantidade média de informação transmitida entre dois pontos em uma rede durante um certo intervalo de tempo (JYOTHI *et al.*, 2016). Ela pode ser um indicador de possíveis gargalos na transmissão de dados de uma rede.

Em relação a algoritmos de roteamento, podemos usar essa métrica para avaliar a qualidade da estratégia de roteamento de um algoritmo quanto a reduzir congestionamentos e melhorar a vazão de dados de uma rede.

3.4.5 Taxa de Transferência Normalizada

É a razão entre a taxa de transferência e a quantidade máxima de dados que podem ser transferidos.

3.5 Algoritmos Genéticos

Algoritmos genéticos empregam técnicas de otimização global. Eles funcionam de forma análoga ao processo de seleção natural na forma como as soluções são codificadas e escolhidas (PACHECO, 2001). Cada possível solução é associada a um valor de *fitness* calculado usando uma função de avaliação que mede sua qualidade. O objetivo é encontrar as soluções que possuam o melhor valor de *fitness* no contexto do problema.

Cada candidato a uma solução no espaço de busca é conhecido como cromossomo ou indivíduo (PACHECO, 2001). O cromossomo é uma estrutura de dados que representa um indivíduo. Ele é dividido em pedaços menores chamados genes que compõe a estrutura de dados. Um exemplo seria um vetor de inteiros, este representa o cromossomo e cada posição nele equivale a um gene.

Os indivíduos são agrupados em um conjunto denominado população (KATO, 2021). Inicialmente, durante a execução de um algoritmo genético, uma população inicial é gerada de forma aleatória seguindo as regras de codificação dos cromossomos no contexto do problema. No decorrer da execução, essa população é modificada para criar uma geração que consiste em indivíduos da geração anteriores mantidos ou alterados. O algoritmo termina quando uma solução ótima é encontrada ou quando o número máximo de gerações atinge o valor alvo.

Uma nova geração é construída aplicando operadores em uma população (KATO, 2021). Os operadores servem para escolher ou alterar indivíduos de uma população em um processo similar ao que acontece na natureza durante a seleção natural. São possíveis operadores: seleção, mutação e *crossover*.

3.5.1 Seleção

Dada uma população, o operador de seleção escolhe um par de cromossomos (KATO, 2021). A decisão varia segundo a implementação, mas, em geral, a escolha ocorre pelo valor de *fitness* dos indivíduos.

3.5.2 Crossover

Esta operação combina dois cromossomos para formar um novo (PACHECO, 2001). O objetivo é unir os genes de dois cromossomos e garantir que as características dos pais passem para a prole. Como os cromossomos são divididos e unidos depende da estrutura de dados utilizada para representá-los.

No caso de cromossomos representados por vetores, uma forma comum de *crossover* é combinar a metade de um vetor com o outro, obtendo assim um novo vetor. Dessa forma pode ser possível criar uma solução que combine as melhores características dos cromossomos pais.

3.5.3 Mutação

A mutação consiste em alterar de forma arbitrária partes da estrutura de dados que representa um cromossomo (KATO, 2021). O objetivo é tentar replicar o processo de mutações genéticas que acontece na natureza. Assim, soluções geradas através da mutação podem conter valores de *fitness* melhores.

Por exemplo, se o cromossomo fosse uma *string* composta por dígitos binários, então a mutação poderia inverter um ou mais dígitos da *string* original seguindo alguma probabilidade.

4 PROCEDIMENTOS METODOLÓGICOS

Nas seções a seguir serão descritas as atividades realizadas para o desenvolvimento deste projeto. De início, será abordado em detalhes o funcionamento do algoritmo proposto e do algoritmo guloso utilizado nos experimentos. As seções seguintes descrevem a implementação dos algoritmos e a configuração do ambiente de testes.

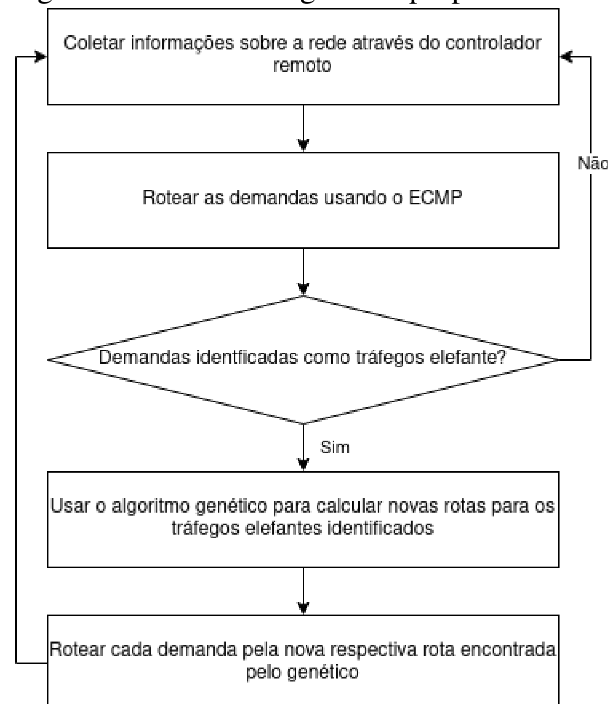
4.1 Algoritmo proposto

Nesta seção será apresentado o algoritmo proposto. Nas subseções a seguir, o objetivo do algoritmo e seu fluxo serão descritos em maiores detalhes.

4.1.1 Visão geral

O algoritmo proposto consiste em um algoritmo genético cujo objetivo é minimizar a utilização dos *links* de uma rede. Ele busca fazer um balanceamento de carga dos *links* presentes para tentar diminuir a ocorrência de congestionamentos. Além disso, esse algoritmo é aplicado sempre que um ou mais tráfegos elefante são detectados em uma rede.

Figura 3 – Fluxo do algoritmo proposto



Fonte: elaborado pelo autor

Os passos do algoritmo proposto são mostrados no fluxograma da Figura 3. Os

pontos principais são descritos como se segue:

1. No controlador de rede, informações sobre a largura de banda dos *links* e sobre o estado da rede são coletadas
2. As demandas na rede são roteadas por padrão usando o ECMP
3. Quando uma ou mais demandas são identificadas como tráfegos elefante, um algoritmo genético é empregado para calcular novas rotas para essas demandas
4. As demandas são roteadas novamente de acordo com a solução do genético

4.1.2 Algoritmo genético

O algoritmo genético recebe uma lista de demandas e gera uma população baseado nelas. Cada demanda passada possui informações como: origem, destino e tamanho. A cada iteração, ele escolhe as soluções que geram o menor valor de maior utilização de *link*. Depois de um certo número de iterações, a melhor solução segundo o critério de busca determinado é a escolhida.

Uma possível solução consiste em uma lista de caminhos que são respectivos a cada demanda passada.

A geração da população e detalhes sobre os operadores usados são descritos nas subseções seguintes.

4.1.2.1 Codificação

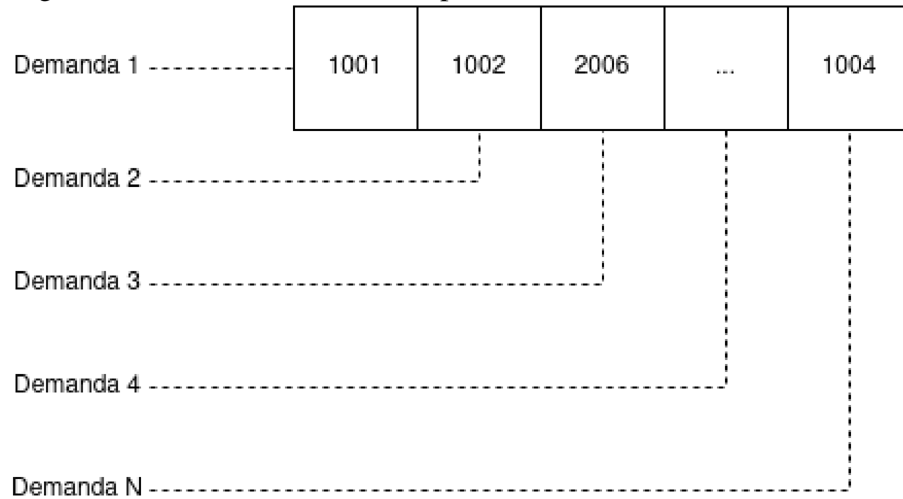
A Figura 7 mostra a representação de um possível cromossomo no algoritmo proposto. Cada solução no algoritmo genético é representada por uma lista de inteiros. Cada posição na lista corresponde a uma demanda passada.

Os elementos da lista são identificadores. Cada um representa um caminho por onde sua respectiva demanda pode ser encaminhada. Dessa forma, no exemplo da Figura 7, se considerássemos esse cromossomo como uma solução final, então a demanda 1 seria encaminhada pelo caminho de identificador 1001.

Os identificadores dos caminhos correspondem a numeração dos *switches core* e de agregação da rede. Sendo que os números que começam com 1 representam *switches core* e 2 os *switches* de agregação.

Os caminhos possíveis não possuem repetição de nós. Cada caminho pode ser obtido sabendo-se a origem e o destino da demanda, e se o identificador do caminho pertence a um

Figura 4 – Cromossomo de exemplo

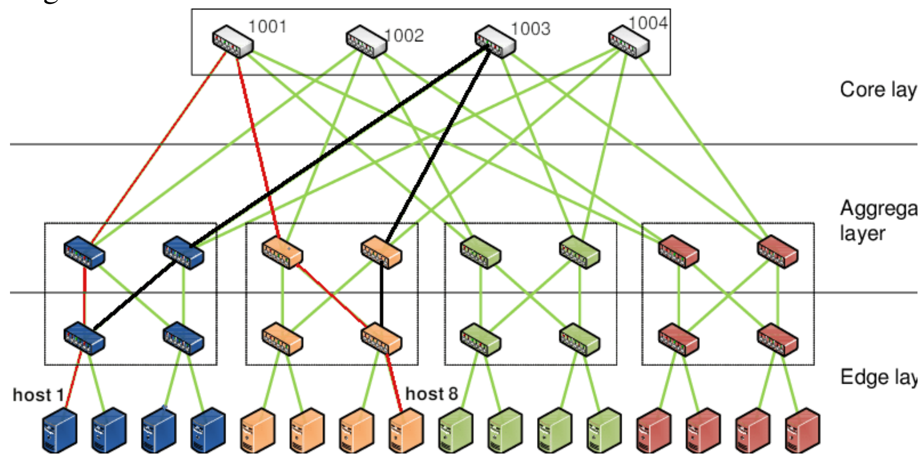


Fonte: elaborado pelo autor

switch core ou de agregação. Então busca-se o menor caminho, entre a origem e o destino, que passa pelo *switch core* ou de agregação associado.

A Figura 5 mostra dois caminhos possíveis para uma demanda entre os *hosts* 1 e 8. O caminho vermelho é representado pelo *switch core* 1001 enquanto que o caminho preto está associado ao identificador 1003.

Figura 5 – Caminhos 1001 e 1003

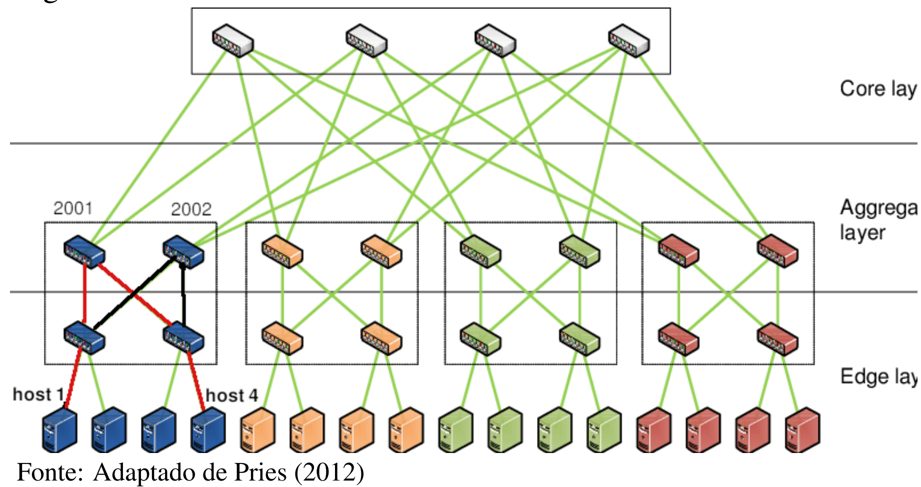


Fonte: Adaptado de Pries (2012)

A Figura 6 mostra os caminhos possíveis para uma demanda dentro de um mesmo *pod*. Os identificadores possíveis para essa demanda, 2001 e 2002, são referentes aos *switches* de agregação presentes no respectivo *pod* que contém os caminhos em destaque.

Inicialmente, quando um indivíduo é criado durante a geração da população, cada demanda entre *pods* é associada a um *switch core*, as demais são associadas a *switches* de agregação.

Figura 6 – Caminhos 2001 e 2002



Fonte: Adaptado de Pries (2012)

Durante a mutação, podemos mudar o caminho de uma demanda ao mudar seu identificador pela numeração de um *switch* do mesmo tipo. No entanto, precisamos respeitar que só podemos trocar um *switch* de agregação por outro que está no mesmo *pod*.

4.1.2.2 Geração da população

A população inicial é gerada de forma pseudoaleatória. O número de indivíduos presentes é igual ao valor do parâmetro respectivo passado durante a inicialização do genético.

Durante a criação de um indivíduo, cada elemento na lista é escolhido de forma arbitrária respeitando se a demanda respectiva é entre *pods* ou não. Dessa forma, se, por exemplo, uma demanda for entre *pods*, o elemento da lista associado deverá ser o identificador de um dos possíveis *switches core*.

4.1.2.3 Função de avaliação

A função de avaliação utilizada pelo algoritmo genético retorna o valor da maior utilização de *link* entre os *links* dos caminhos das demandas no cromossomo. Para isso, ela considera o estado da *fat-tree* após o roteamento das demandas presentes no cromossomo em relação aos caminhos associados. Então, identifica o *link* com a maior utilização e retorna esse valor.

4.1.2.4 Operação de seleção

Os indivíduos que passarão para a próxima geração são escolhidos segundo a probabilidade da Fórmula 4.2. Dado um cromossomo c_i onde N é o tamanho da população e F é a

função de avaliação, a probabilidade de c_i ser escolhido é $P(c_i)$.

Durante a seleção o objetivo é escolher os indivíduos com o menor número de maior utilização de *link*. Por isso, a probabilidade $p(c_i)$ de um cromossomo c_i ser escolhido é igual ao valor normalizado $q(c_i)$.

$$q(c_i) = 1 - \frac{F(c_i)}{\sum_{j=1}^{n} F(c_j)} \tag{4.1}$$

$$p(c_i) = \frac{q(c_i)}{\sum_{j=1}^{n} q(c_j)} \tag{4.2}$$

4.1.2.5 Operação de mutação

Na operação de mutação, elementos da lista que compõe um cromossomo são alterados de forma aleatória. A chance de um indivíduo sofrer mutação segue a probabilidade passada como parâmetro para o algoritmo genético.

Os elementos alterados nos cromossomos são substituídos por valores equivalentes. Por isso, se o caminho atual associado a uma demanda for representado por um *switch core* no cromossomo, durante a mutação, esse elemento poderá ser substituído por outro *switch core*, o que no contexto do algoritmo proposto significaria associar um novo caminho à demanda.

4.1.2.6 Operação de crossover

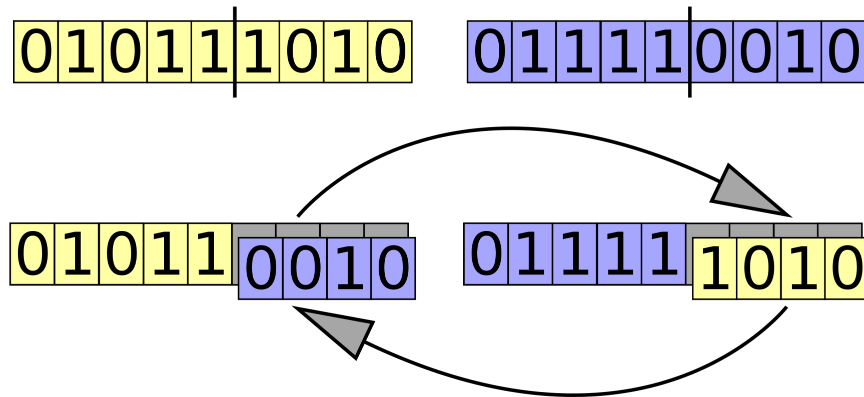
A operação de *crossover* recebe dois cromossomos, segmenta eles em uma posição aleatória e une cada pedaço criado com sua parte correspondente no cromossomo oposto. Desse processo são formados dois novos indivíduos que combinam as características dos cromossomos originais como mostra a Figura X.

Essa operação é utilizada apenas quando o número de elementos nos indivíduos de entrada é maior que 1.

4.2 Algoritmo guloso

Além do Hedera e do ECMP, o trabalho proposto será comparado a um algoritmo guloso. O objetivo é comparar a qualidade dos resultados do algoritmo proposto contra um algoritmo de estratégia simples, além do ECMP.

Figura 7 – Operação de crossover



Fonte: RECOMBINAÇÃO (COMPUTAÇÃO EVOLUTIVA) (2019)

Sempre que um tráfego elefante é identificado, a heurística gulosa escolhe de forma arbitrária um número N de caminhos, dando preferência aos caminhos mais curtos. Então a demanda é encaminhada pelo caminho com a maior capacidade disponível.

O objetivo do guloso é tentar reduzir congestionamentos na rede ao rotear os tráfegos elefante pelos caminhos mais livres.

O Algoritmo 1 apresenta os passos da heurística gulosa. Como entrada, o algoritmo recebe um número N que representa a quantidade de demandas para as quais deseja-se calcular novos caminhos; um valor K que consiste na quantidade de caminhos a se considerar para cada demanda; listas src e dst que contém, respectivamente, a origem e o destino de cada demanda. Como saída, o algoritmo retorna uma lista P com os caminhos escolhidos para cada demanda passada.

Inicialmente, durante a execução do Algoritmo 1, a lista P é inicializada como uma lista vazia. Então, para cada demanda i, são inicializadas 3 variáveis, são elas: $bestPath$ que representa o melhor caminho associado a demanda atual, $bestPathFreeCapacity$ que é a capacidade disponível de $bestPath$ e $paths$ que consiste em uma lista com os K menores caminhos possíveis, em número de nós, para a demanda i.

Para cada caminho $path$ em $paths$ no Algoritmo 1, temos $bestLink$ como o $link$ com a menor capacidade atual para o caminho $path$ e $bestLinkFreeCapacity$ como a capacidade de $bestLink$. De início, como $bestPath$ é um valor nulo, $bestLinkFreeCapacity$ é inicializado com $MAX_CAPACITY+1$, pois a heurística gulosa dá preferência aos *links* com a menor capacidade disponível para encontrar o caminho com a maior capacidade disponível.

Para cada *link* $pathLink$ em $path$, o Algoritmo 1 salva a capacidade disponível de

pathLink em *linkFreeCapacity*, então compara a capacidade de *pathLink* com *bestLink*. Caso a capacidade de *pathLink* for menor ou igual a *bestLink*, então *bestLink* recebe *pathLink* e *bestLinkFreeCapacity* recebe *linkFreeCapacity*. Por fim, a capacidade disponível do caminho *path* será *bestLinkFreeCapacity*.

Após passar por cada *link* do caminho *path*, o Algoritmo 1 verifica se a capacidade de *path*, representada por *bestLinkFreeCapacity*, é maior que a capacidade disponível do melhor caminho até o momento, *bestPath*. Se for maior, então *bestPath* recebe *path* e *bestPathFreeCapacity* recebe *bestLinkFreeCapacity*. Dessa forma, *path* passa a ser o melhor caminho atual para a demanda i.

Após avaliar cada caminho *path*, o Algoritmo 1 salva o melhor caminho encontrado para a demanda i, *bestPath*, em $P[i]$. Por fim, o algoritmo retorna a lista P com os novos caminhos para cada demanda passada.

Algoritmo 1: Guloso

```
entrada :
N: número de demandas.
K: número de caminhos a se considerar para cada demanda.
src[i]: origem da i-ésima demanda.
dst[i]: destino da i-ésima demanda.
saída    : uma lista P com os caminhos de cada demanda.
P ← {}
for i ← 1 to N do
    bestPath ← nil
    bestPathFreeCapacity ← 0
    paths ← K menores caminhos entre src[i] e dst[i]
    foreach path in paths do
        bestLink ← nil
        bestLinkFreeCapacity ← MAX_CAPACITY + 1
        foreach pathLink in path do
            linkFreeCapacity ← capacidade disponível do link pathLink
            if linkFreeCapacity ≤ bestLinkFreeCapacity then
                bestLink ← pathLink
                bestLinkFreeCapacity ← linkFreeCapacity
            end
        end
        if bestLinkFreeCapacity > bestPathFreeCapacity then
            bestPath ← path
            bestPathFreeCapacity ← bestLinkFreeCapacity
        end
    end
    P[i] ← bestPath
end
return P
```

Fonte: Elaborado pelo autor.

4.3 Implementação dos algoritmos utilizados

Os algoritmos escolhidos para comparação com o genético proposto foram o Hedera, o ECMP e o algoritmo guloso.

A linguagem utilizada no projeto foi Python. Os algoritmos usados neste trabalho foram implementados na aplicação desenvolvida por Huangmachi (2017), exceto o Hedera e o ECMP, pois esses já possuem implementação própria na aplicação utilizada.

A aplicação de Huangmachi (2017) é responsável por criar redes virtuais emuladas definidas por software com topologia *fat-tree* utilizando a ferramenta Mininet (MININET, 2018), gerar os tráfegos de teste e configurar o controlador remoto Ryu (RYU, 2018).

4.4 Ambiente de testes

Os experimentos foram realizados em *fat-trees* com 4 e 8 *pods* usando *links* com capacidade de 100 Mbps. Os algoritmos utilizados foram executados individualmente durante 60 segundos para cada padrão tráfego escolhido. A Tabela 1 apresenta a configuração da máquina utilizada nos testes.

Tabela 1: Configuração da máquina de testes

Sistema operacional	Ubuntu 16.04.7 LTS
Quantidade de memória ram	32GB
Número de Cores	8
Número de Threads	16

Fonte: o autor.

Os padrões de tráfego escolhidos para o experimento são mostrados na Tabela 2. Cada padrão difere quanto a proporção de tráfego gerado dentro de um mesmo *switch* de borda, entre diferentes *switches* de borda dentro de um mesmo *pod* e entre tráfegos entre diferentes *pods*. Além disso, padrões de tráfego do tipo *random* possuem uma proporção de tráfego aleatória.

Tabela 2: Padrões de tráfego usados

Tráfego	Mesmo *switch* de borda	Diferentes *switches* de borda no mesmo pod	Diferentes pods
stag_0.2_0.3	20%	30%	50%
stag_0.4_0.3	40%	30%	30%
stag_0.5_0.3	60%	20%	20%

Fonte: o autor.

4.5 Métricas para avaliação

A Tabela 3 mostra as métricas utilizadas para comparar os algoritmos do experimento. Além disso, ela contém a descrição de cada métrica.

Tabela 3: Métricas utilizadas no experimento

Métrica	**Descrição**
Taxa de transferência	Quantidade de dados transferidos na rede, considerando todos os *hosts*, durante um certo intervalo de tempo.
Taxa de transferência Normalizada	Razão entre a taxa de transferência e a soma da capacidade dos *links* dos *hosts* na rede.
Taxa de perda de pacotes	Razão entre número pacotes de entregues e de pacotes transmitidos.
Latência Bidirecional	Tempo para enviar um pacote e receber a confirmação de sua entrega.

Fonte: o autor.

5 RESULTADOS

Neste capítulo serão apresentados os resultados deste trabalho para o algoritmo genético proposto e sua comparação com os algoritmos escolhidos.

O algoritmo genético foi testado com diferentes parâmetros. As configurações testadas são variações dos parâmetros escolhidos no trabalho de (HONGHUI *et al.*, 2020). O objetivo foi escolher 10 parâmetros e testar para diferentes padrões de tráfego e topologias, *fat-trees* com 4 e 8 *pods*. As combinações que geraram os melhores resultados em média para as métricas escolhidas foram destacadas e comparadas aos algoritmos escolhidos, que seriam: o Hedera, o ECMP e a heurística gulosa.

5.1 Resultados do algoritmo genético para *fat-trees* com 4 pods

As tabelas a seguir mostram os resultados do algoritmo genético para os diferentes padrões de tráfego escolhidos, usando a topologia *fat-tree* com 4 *pods*. Além disso, cada tabela inclui resultados para cada parâmetro do genético testado.

Tabela 4: Resultados do genético para *fat-tree* com 4 *pods* e tráfego stag_0.2_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	707.18	0.442	0.00362	53.91
100	25	0.6	0.25	684.43	0.428	0.00143	29.37
100	10	0.6	0.25	662.56	0.414	0.00143	42.32
50	50	0.6	0.25	680.42	0.425	0.00061	33.63
50	25	0.6	0.25	679.10	0.424	0.00092	25.45
25	**50**	**0.6**	**0.25**	**713.78**	**0.446**	**0.00026**	**26.03**
25	25	0.6	0.25	669.66	0.419	0.00061	57.06
100	50	0.8	0.2	671.38	0.420	0.00158	39.85
50	10	0.8	0.1	702.95	0.439	0.00241	43.54
25	25	0.8	0.1	688.76	0.430	0.00105	58.77

Fonte: o autor.

A Tabela 4 mostra os resultados do genético para o padrão de tráfego stag_0.2_0.3. Entre os parâmetros testados, podemos observar que de forma geral a melhor configuração de parâmetros foi 25 gerações, população de 50 indivíduos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25. Esses parâmetros geraram a maior taxa de transferência e a menor perda de pacotes. Apesar dessa configuração não ter alcançado a menor latência, possui uma das menores.

Como mostra a Tabela 5, no geral, o melhor conjunto de parâmetros para o tráfego stag_0.4_0.3 foi 50 gerações, população de 10 cromossomos, taxa de *crossover* de 0.8 e taxa de mutação de 0.1. Esses parâmetros produziram a menor latência e 0 perda de pacotes.

Tabela 5: Resultados do genético para *fat-tree* e com 4 *pods* e tráfego stag_0.4_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	969.47	0.606	0.00016	25.83
100	25	0.6	0.25	960.12	0.600	0.00028	54.82
100	10	0.6	0.25	960.13	0.600	0.00012	19.75
50	**50**	**0.6**	**0.25**	**948.99**	**0.593**	**0.00000**	**12.94**
50	25	0.6	0.25	932.98	0.583	0.00024	21.22
25	50	0.6	0.25	950.89	0.594	0.00060	102.85
25	**25**	**0.6**	**0.25**	**957.11**	**0.598**	**0.00000**	**14.05**
100	50	0.8	0.2	951.81	0.595	0.00044	38.31
50	**10**	**0.8**	**0.1**	**946.55**	**0.592**	**0.00000**	**2.72**
25	25	0.8	0.1	958.83	0.599	0.00063	84.79

Fonte: o autor.

Tabela 6: Resultados do genético para *fat-trees* com 4 *pods* e tráfego stag_0.5_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	924.55	0.578	0.00063	52.64
100	**25**	**0.6**	**0.25**	**921.70**	**0.576**	**0.00008**	**30.10**
100	10	0.6	0.25	913.37	0.571	0.00147	34.49
50	50	0.6	0.25	919.47	0.575	0.00048	37.59
50	25	0.6	0.25	916.37	0.573	0.00024	44.51
25	**50**	**0.6**	**0.25**	**930.27**	**0.581**	**0.00067**	**62.99**
25	25	0.6	0.25	913.21	0.571	0.00012	40.34
100	50	0.8	0.2	910.64	0.569	0.00036	38.55
50	10	0.8	0.1	923.47	0.577	0.00063	32.38
25	25	0.8	0.1	913.47	0.571	0.00056	42.93

Fonte: o autor.

De forma geral, o melhor conjunto de parâmetros para o tráfego stag_0.5_0.3 foi 100 gerações, 25 indivíduos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25, conforme apresentado na Tabela 6.

Tabela 7: Resultados do genético para *fat-trees* com 4 *pods* e tráfego random1

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	729.32	0.456	0.00167	39.39
100	25	0.6	0.25	725.92	0.454	0.00069	38.59
100	10	0.6	0.25	715.66	0.447	0.00222	42.96
50	50	0.6	0.25	730.07	0.456	0.00292	43.96
50	25	0.6	0.25	698.20	0.436	0.00306	66.68
25	50	0.6	0.25	716.19	0.448	0.00042	51.84
25	**25**	**0.6**	**0.25**	**751.89**	**0.470**	**0.00069**	**47.41**
100	50	0.8	0.2	719.32	0.450	0.00181	60.62
50	10	0.8	0.1	748.56	0.468	0.00333	54.62
25	**25**	**0.8**	**0.1**	**735.38**	**0.460**	**0.00000**	**50.83**

Fonte: o autor.

A melhor configuração de parâmetros para o tráfego random1, segundo a Tabela 7, foi 25 gerações, 25 cromossomos, taxa de *crossover* de 0.8 e taxa de mutação de 0.1. Isso considerando que esse conjunto de parâmetros apresentou a menor perda de pacotes. Apesar

disso, a configuração 25 gerações, 25 cromossomos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25, apresentou uma taxa de transferência mais alta e uma menor latência.

Tabela 8: Resultados do genético para *fat-trees* com 4 *pods* e tráfego random2

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	635.94	0.397	0.00333	63.53
100	25	0.6	0.25	661.35	0.413	0.00075	29.43
100	10	0.6	0.25	662.32	0.414	0.00242	44.77
50	50	0.6	0.25	643.98	0.402	0.00192	68.06
50	25	0.6	0.25	647.29	0.405	0.00083	35.12
25	**50**	**0.6**	**0.25**	**669.07**	**0.418**	**0.00000**	**29.97**
25	25	0.6	0.25	648.97	0.406	0.00133	31.07
100	50	0.8	0.2	648.69	0.405	0.00158	40.60
50	10	0.8	0.1	661.85	0.414	0.00300	42.80
25	25	0.8	0.1	655.36	0.410	0.00117	35.39

Fonte: o autor.

Na Tabela 8 podemos ver que o melhor resultado para o tráfego random2 foi a combinação de parâmetros 25 gerações, população de 100 indivíduos, taxa de *crossover* de 0.6 e taxa de mutação de 0.6. Essa configuração apresentou a melhor taxa de transferência e perda de pacotes. Além disso, ela conseguiu manter uma latência baixa comparada aos outros parâmetros.

5.2 Resultados do algoritmo genético para *fat-trees* com 8 *pods*

As tabelas a seguir mostram os resultados do algoritmo genético para *fat-trees* com 8 *pods*. Os parâmetros utilizados foram os mesmos dos testes feitos com as *fat-trees* com 4 *pods*. O objetivo desses experimentos foi mostrar como os resultados do algoritmo proposto são afetados conforme o número de *switches* e *hosts* da rede aumenta.

Tabela 9: Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.2_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	5683.38	0.444	0.00274	171.86
100	25	0.6	0.25	5686.98	0.444	0.00199	172.59
100	10	0.6	0.25	5660.29	0.442	0.00261	187.01
50	50	0.6	0.25	5724.66	0.447	0.00250	186.98
50	25	0.6	0.25	5800.36	0.453	0.00261	165.97
25	50	0.6	0.25	5711.00	0.446	0.00285	166.40
25	25	0.6	0.25	5652.58	0.442	0.00240	175.92
100	**50**	**0.8**	**0.2**	**5767.44**	**0.451**	**0.00200**	**162.42**
50	10	0.8	0.1	5557.63	0.434	0.00248	179.94
25	25	0.8	0.1	5582.43	0.436	0.00225	182.78

Fonte: o autor.

A Tabela 9 mostra que para o padrão de tráfego stag_0.2_0.3, o melhor conjunto de

parâmetros foi 100 gerações, 50 cromossomos, taxa de *crossover* de 0.8 e taxa de mutação de 0.2. Isso porque esse grupo de parâmetros gerou uma das menores perdas de pacotes e a menor latência. Além disso, ele apresentou uma das maiores taxas de transferência.

Tabela 10: Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.4_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	6457.15	0.504	0.00177	138.06
100	25	0.6	0.25	6439.93	0.503	0.00187	125.27
100	**10**	**0.6**	**0.25**	**6352.32**	**0.496**	**0.00109**	**116.14**
50	50	0.6	0.25	6389.71	0.499	0.00173	141.43
50	25	0.6	0.25	6366.11	0.497	0.00165	144.10
25	50	0.6	0.25	6124.92	0.479	0.00156	141.10
25	25	0.6	0.25	6414.75	0.501	0.00186	125.61
100	50	0.8	0.2	6468.01	0.505	0.00178	127.68
50	10	0.8	0.1	6483.79	0.507	0.00181	149.23
25	25	0.8	0.1	6310.26	0.493	0.00161	116.33

Fonte: o autor.

Na Tabela 10 temos que a melhor configuração de parâmetros para o padrão de tráfego stag_0.4_0.3 foi 100 gerações, 50 indivíduos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25. Esses parâmetros apresentaram a menor perda de pacotes e latência.

Tabela 11: Resultados do genético para *fat-trees* com 8 *pods* e tráfego stag_0.5_0.3

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	7492.84	0.585	0.00110	106.83
100	25	0.6	0.25	7411.56	0.579	0.00064	92.67
100	**10**	**0.6**	**0.25**	**7476.82**	**0.584**	**0.00058**	**85.27**
50	50	0.6	0.25	7363.58	0.575	0.00102	95.54
50	25	0.6	0.25	7503.49	0.586	0.00087	79.09
25	50	0.6	0.25	7390.31	0.577	0.00077	113.74
25	25	0.6	0.25	7539.23	0.589	0.00094	112.81
100	50	0.8	0.2	7407.19	0.579	0.00109	99.32
50	10	0.8	0.1	7515.04	0.587	0.00106	81.25
25	25	0.8	0.1	7570.21	0.591	0.00092	86.94

Fonte: o autor.

Para o padrão de tráfego stag_0.5_0.3 o conjunto de parâmetros com os melhores resultados de forma geral foi 100 gerações, 10 cromossomos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25; como pode ser visto na Tabela 11. Essa combinação resultou em uma latência reduzida e uma baixa taxa de perda de pacotes.

Na Tabela 12 temos que para o padrão do tipo random1 a melhor combinação de parâmetros foi 100 gerações, 10 indivíduos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25. Essa configuração produziu a menor latência e a menor taxa de perda de pacotes.

Tabela 12: Resultados do genético para *fat-trees* com 8 *pods* e tráfego random1

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	3563.72	0.278	0.00258	175.11
100	25	0.6	0.25	3574.99	0.279	0.00263	186.80
100	**10**	**0.6**	**0.25**	**3424.76**	**0.268**	**0.00229**	**153.93**
50	50	0.6	0.25	3604.54	0.282	0.00298	202.37
50	25	0.6	0.25	3807.58	0.297	0.00320	190.15
25	50	0.6	0.25	3438.39	0.269	0.00231	169.73
25	25	0.6	0.25	3501.66	0.274	0.00279	172.65
100	50	0.8	0.2	3553.40	0.278	0.00239	199.05
50	10	0.8	0.1	3528.35	0.276	0.00260	148.43
25	25	0.8	0.1	3640.65	0.284	0.00280	181.94

Fonte: o autor.

Tabela 13: Resultados do genético para *fat-trees* com 8 *pods* e tráfego random2

Gerações	Pop.	Crossover	Mutação	Transf.	Transf. Normalizada	Perda de Pacotes	Lat. Bidirecional
100	100	0.6	0.25	3741.23	0.292	0.00277	187.80
100	25	0.6	0.25	3615.95	0.282	0.00219	165.13
100	**10**	**0.6**	**0.25**	**3639.02**	**0.284**	**0.00214**	**151.16**
50	50	0.6	0.25	3604.27	0.282	0.00251	154.86
50	25	0.6	0.25	3696.84	0.289	0.00309	168.20
25	50	0.6	0.25	3725.10	0.291	0.00243	153.74
25	25	0.6	0.25	3583.91	0.280	0.00227	165.14
100	50	0.8	0.2	3753.89	0.293	0.00230	168.32
50	10	0.8	0.1	3808.23	0.298	0.00278	167.20
25	25	0.8	0.1	3752.67	0.293	0.00265	166.29

Fonte: o autor.

A Tabela 13 mostra que para o padrão de tráfego random2 a melhor configuração de parâmetros foi 100 gerações, 10 cromossomos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25. Esse grupo de parâmetros, apesar de não apresentar uma das maiores taxas de transferência, possui a menor taxa de perdas de pacote e a melhor latência bidirecional.

Para as *fat-trees* com 8 *pods*, segundo os resultados apresentados, podemos observar que a melhor configuração de parâmetros, em geral, foi 100 gerações, 10 cromossomos, taxa de *crossover* de 0.6 e taxa de mutação de 0.25. Uma combinação diferente do que foi visto nos resultados para as *fat-trees* de 4 *pods*.

5.3 Comparação com os algoritmos escolhidos para *fat-trees* com 4 *pods*

Os gráficos a seguir mostram comparações entre o algoritmo proposto e os algoritmos escolhidos, que são: o Hedera, o ECMP e o algoritmo guloso. A topologia alvo é a *fat-tree* de 4 *pods*.

Os resultados do algoritmo genético presentes nas comparações correspondem aos

resultados gerados com os parâmetros mostrados na Tabela X. Esses parâmetros foram escolhidos por apresentarem bons resultados em relação as outras configurações testadas.

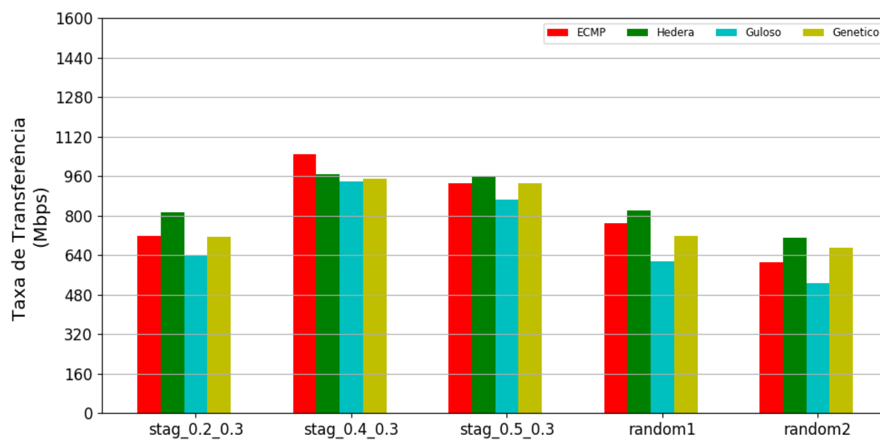
Tabela 14: Parâmetros do algoritmo genético para *fat-trees* com 4 *pods*

Parâmetro	Valor
Gerações	25
População	50
Probabilidade de crossover	0.6
Probabilidade de mutação	0.25

Fonte: o autor.

Nas Figuras 8 e 9 temos uma comparação da taxa de transferência entre os algoritmos escolhidos. O algoritmo guloso apresentou os piores valores. O algoritmo genético se manteve próximo do ECMP para os padrões stag_0.5_0.3 e stag_0.2_0.3. Além disso, o genético apresentou uma taxa de transferência superior ao ECMP para o tráfego random2.

Figura 8: Taxa de Transferência em Mbps - Fat-Tree com *4* pods

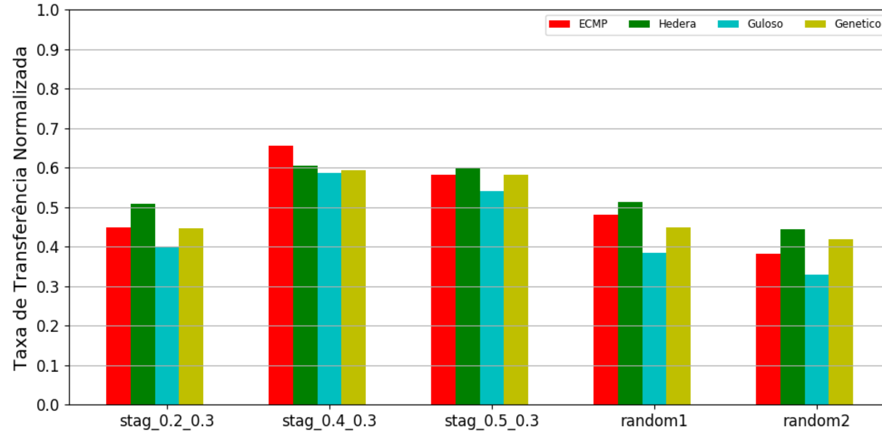


Fonte: elaborado pelo autor

Na Figura 10 temos uma comparação dos algoritmos escolhidos em relação à taxa de perda de pacotes. Como esperado, o ECMP apresentou os piores resultados, dado sua estratégia de roteamento (ZHANG *et al.*, 2015). O algoritmo genético gerou a menor taxa de perda de pacotes, quanto aos outros algoritmos nos tráfegos stag_0.2_0.3, random1 e random2.

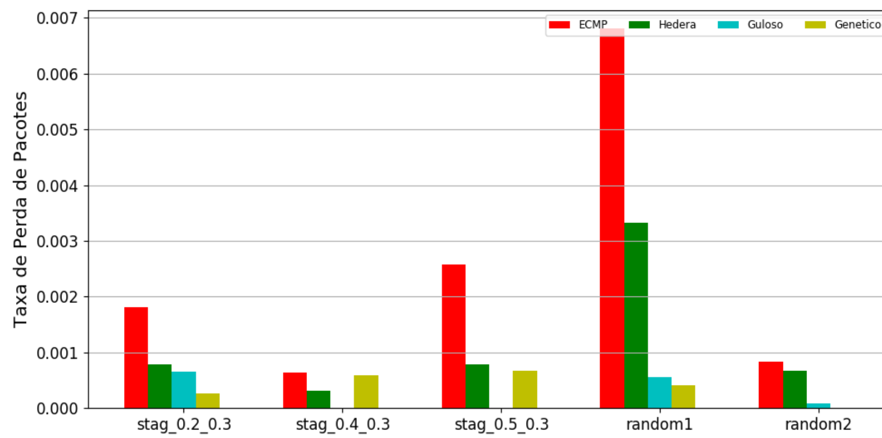
A Figura 11 mostra as latências dos algoritmos testados. O algoritmo genético apresentou os melhores resultados nos padrões de tráfego stag_0.2_0.3, random1 e random2. O Guloso se manteve com latências mais baixas que as do Hedera e do ECMP.

Figura 9: Taxa de Transferência Normalizada - Fat-Tree com *4* pods



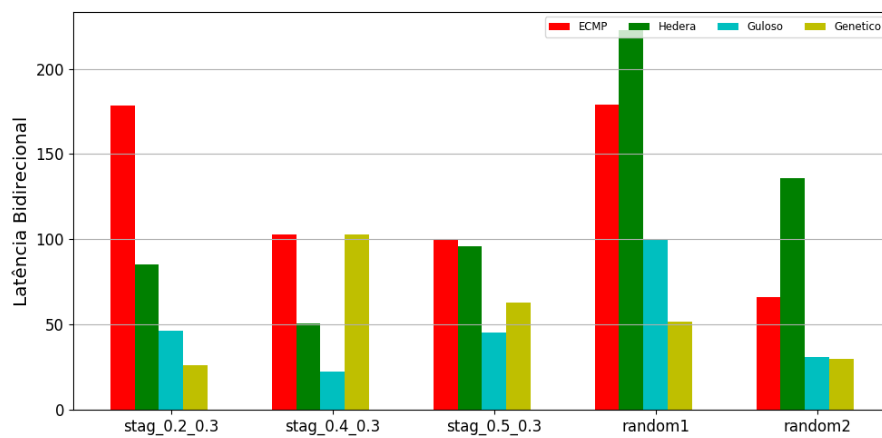
Fonte: elaborado pelo autor

Figura 10: Taxa de Perda de Pacotes - Fat-Tree com 4 *pods*



Fonte: elaborado pelo autor

Figura 11: Latência Bidirecional



Fonte: elaborado pelo autor

5.4 Comparação com os algoritmos escolhidos para *fat-trees* com 8 pods

Os gráficos seguintes mostram comparações entre o algoritmo proposto e os algoritmos escolhidos usando *fat-trees* com 8 *pods*. O objetivo dessas comparações é mostrar como o algoritmo genético se comporta quando o tamanho da rede aumenta.

Os resultados do genético utilizados foram gerados usando os parâmetros da Tabela 15. Essa configuração foi escolhida porque se destacou em relação aos outros parâmetros testados para as *fat-trees* com 8 *pods*.

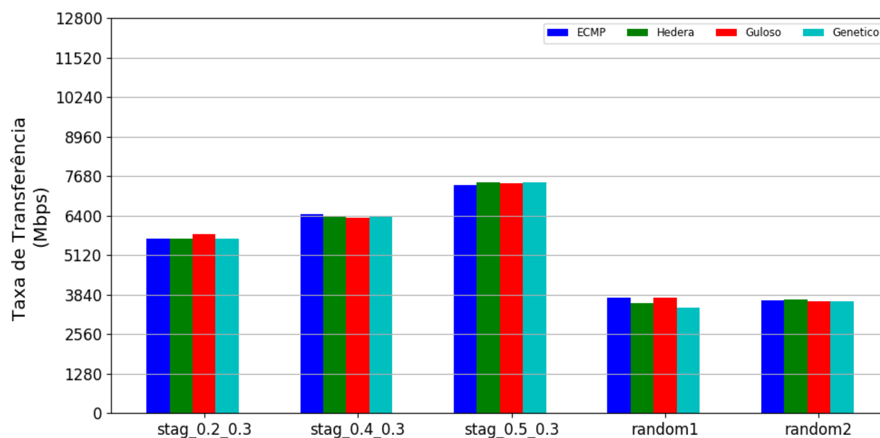
Tabela 15: Parâmetros do algoritmo genético para *fat-trees* com 8 *pods*

Parâmetro	Valor
Gerações	100
População	10
Probabilidade de crossover	0.6
Probabilidade de mutação	0.25

Fonte: o autor.

Nas Figuras 12 e 13 vemos que a taxa de transferência dos algoritmos testados se manteve próxima. Apesar disso, o algoritmo genético apresentou uma taxa de transferência inferior ao outros algoritmos em relação ao padrão de tráfego random1.

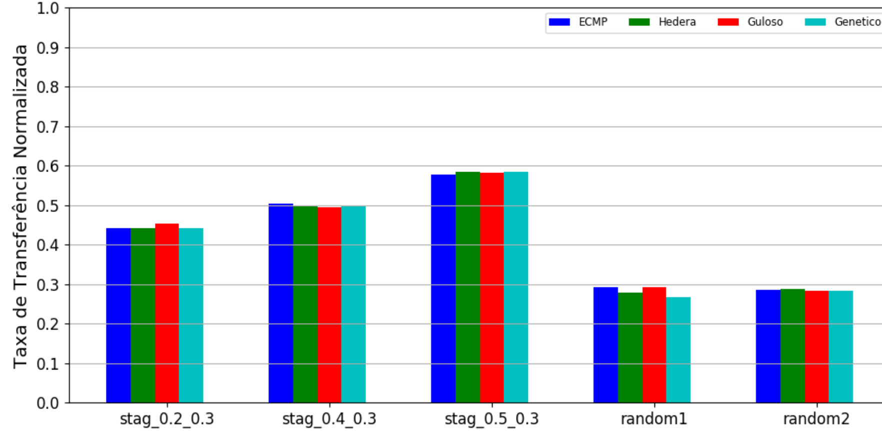
Figura 12: Taxa de Transferência em Mbps



Fonte: elaborado pelo autor

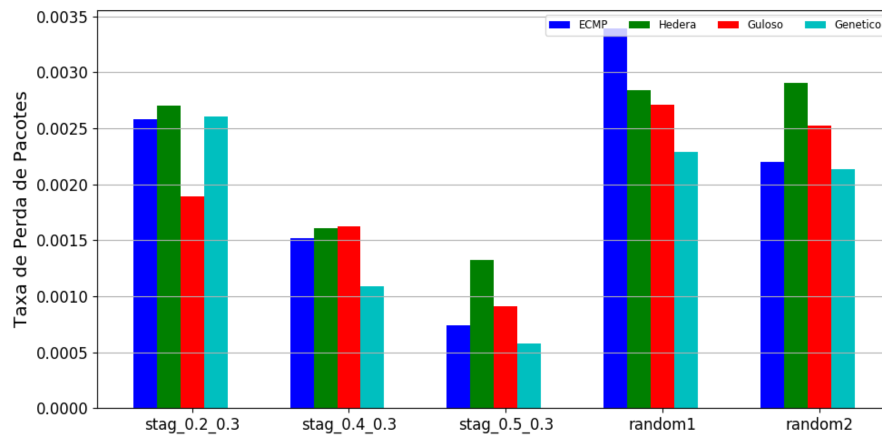
Quanto a taxa de perda de pacotes, Figura 14, o algoritmo genético apresentou os melhores valores entre os algoritmos testados quanto aos padrões de tráfego stag_0.4_0.3, stag_0.5_0.3, random1 e random2. Além disso, o algoritmo guloso também mostrou valores mais baixos que o Hedera e o ECMP.

Figura 13: Taxa de Transferência Normalizada



Fonte: elaborado pelo autor

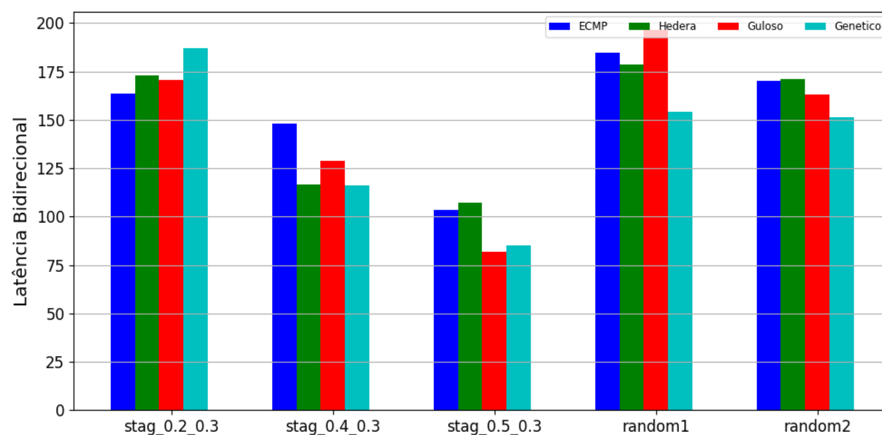
Figura 14: Taxa de Perda de Pacotes



Fonte: elaborado pelo autor

Na Figura 15 temos que o algoritmo genético obteve os melhores resultados para os tráfegos random1 e random2. Ademais, ele apresentou uma latência semelhante ao Hedera em relação ao tráfego stag_0.4_0.3. No entanto, sua latência foi a pior no tráfego stag_0.2_0.3.

Figura 15: Latência Bidirecional



Fonte: elaborado pelo autor

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi apresentado um algoritmo genético para o roteamento de tráfegos elefante. Esse algoritmo identifica demandas que se caracterizam como tráfegos elefante e tenta roteá-los para a minimizar a utilização dos *links* de uma rede. Seu objetivo é reduzir congestionamentos em rede ao tratar especificamente desses tipos de demanda.

O algoritmo proposto e a heurística gulosa foram implementados na aplicação desenvolvida por (HUANGMACHI, 2017). O trabalho usou o software Mininet (MININET, 2018) para simular redes virtuais com arquitetura SDN e o controlador de redes Ryu (RYU, 2018).

O algoritmo genético foi testado com diferentes parâmetros para identificar quais combinações rendem os melhores resultados de acordo com as métricas escolhidas. Percebeu-se que a melhor configuração de parâmetros variou de acordo com a topologia utilizada, *fat-trees* com 4 e 8 *pods*.

O genético foi comparado aos algoritmos ECMP, Hedera e a uma heurística gulosa. Ele apresentou uma redução na taxa de perda de pacotes e na latência bidirecional em relação aos outros algoritmos testados. Além disso, ele mostrou possuir uma taxa de transferência próxima aos demais.

Em relação a trabalhos futuros, uma possibilidade seria investigar possíveis otimizações na implementação da função de avaliação do genético. Ademais, outros objetivos poderiam ser o de testar o genético com outros parâmetros e utilizar *fat-trees* com mais de 8 *pods*.

REFERÊNCIAS

ABHIJIT, B.; ANWAR, H. M. A survey of fat – tree network – on – chip topology. In: **International Journal of Scientific & Technology Research**. [*S. l.*: *s. n.*], 2019. p. 984–980.

AL-FARES, M.; RADHAKRISHNAN, S.; RAGHAVAN, B.; HUANG, N.; VAHDAT, A. Hedera: dynamic flow scheduling for data center networks. In: **NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation**. 2560 Ninth St. Suite 215 Berkeley, CA, United States: USENIX Association, 2010. p. 19.

Alqahtani, J.; Hamdaoui, B. Rethinking fat-tree topology design for cloud data centers. In: **2018 IEEE Global Communications Conference (GLOBECOM)**. [*S. l.*: *s. n.*], 2018. p. 1–6.

CHHABRA, A.; KIRAN, M. Classifying elephant and mice flows in high-speed scientific networks. **Proc. INDIS**, p. 1–8, 2017.

DEHURY, C. What is Fat Tree and how to construct it in 4-steps ?. **Computer Inquisitive**. Balangir, 28 abr. 2017. Disponível em: https://blogchinmaya.blogspot.com/2017/04/what-is-fat-tree-and-how-to-construct.html. Acesso em: 3 mar. 2022.

EISSA, H. A.; BOZED, K. A.; YOUNIS, H. Software defined networking. In: **2019 19th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)**. [*S. l.*: *s. n.*], 2019. p. 620–625.

HAGIWARA, T.; MAJIMA, H.; MATSUDA, T.; YAMAMOTO, M. Impact of round trip delay self-similarity on tcp performance. In: **Proceedings Tenth International Conference on Computer Communications and Networks (Cat. No.01EX495)**. [*S. l.*: *s. n.*], 2001. p. 166–171.

Hassidim, A.; Raz, D.; Segalov, M.; Shaqed, A. Network utilization: The flow view. In: **2013 Proceedings IEEE INFOCOM**. [*S. l.*: *s. n.*], 2013. p. 1429–1437.

HONGHUI, L.; HAILIANG, L.; XUELIANG, F. An optimal and dynamic elephant flow scheduling for sdn-based data center networks. **Journal of Intelligent Fuzzy Systems**, IOS Press, v. 408, n. 1, p. 247–255, 2020.

HUANGMACHI. **Exp_efattree is an experiment to compare the performance of EFattree with ECMP, PureSDN and Hedera**. 2017. Disponível em: https://github.com/Huangmachi/exp_EFattree. Acesso em: 25 jan. 2022.

IWAMA, K.; MIYAZAKI, S. A survey of the stable marriage problem and its variants. In: **International Conference on Informatics Education and Research for Knowledge-Circulating Society (icks 2008)**. Kyoto, Japan: IEEE, 2008. p. 131–136.

JYOTHI, S. A.; SINGLA, A.; GODFREY, P. B.; KOLLA, A. Measuring and understanding throughput of network topologies. In: **SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [*S. l.*: *s. n.*], 2016. p. 761–772.

KATO, R. **Algoritmos Genéticos**. 2021. Disponível em: https://bioinfo.com.br/algoritmos-geneticos/. Acesso em: 23 jan. 2022.

LEE, Y. L.; LOO, J.; CHUAH, T. C. Chapter 24 - modeling and performance evaluation of resource allocation for lte femtocell networks. In: OBAIDAT, M. S.; NICOPOLITIDIS, P.; ZARAI, F. (Ed.). **Modeling and Simulation of Computer Networks and Systems**. Boston: Morgan Kaufmann, 2015. p. 683–716. ISBN 978-0-12-800887-4. Disponível em: https://www.sciencedirect.com/science/article/pii/B9780128008874000249.

LEISERSON, C. Fat-trees: Universal networks for hardware-efficient supercomputing. **IEEE Transactions on Computers**, C-34, p. 892–901, 1985.

MININET. **Introduction to Mininet · mininet/mininet Wiki**. 2018. Disponível em: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet. Acesso em: 27 set. 2020.

MORI, T.; UCHIDA, M.; KAWAHARA, R.; PAN, J.; GOTO, S. Identifying elephant flows through periodically sampled packets. In: **Proceedings of the 4th ACM SIGCOMM conference on Internet measurement**. [*S. l.*: *s. n.*], 2004. p. 115–120.

PACHECO, M. A. C. **Algoritmos genéticos**: princípios e aplicações. 2001. Disponível em: http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf. Acesso em: 23 jan. 2022.

PRIES, R. **Fig. 5. Fat-tree data center architecture.** 2012. Disponível em: https://www.researchgate.net/figure/Fat-tree-data-center-architecture_fig5_220018693. Acesso em: 24 fev. 2022.

RAOUF, O. A.; ASKR, H. Acosdn-ant colony optimization algorithm for dynamic routing in software defined networking. In: **2019 14th International Conference on Computer Engineering and Systems (ICCES)**. [*S. l.*: *s. n.*], 2019. p. 141–148.

RECOMBINAÇÃO (COMPUTAÇÃO EVOLUTIVA). In: **WIKIPÉDIA, a enciclopédia livre**. Flórida: Wikimedia Foundation, 2019. Disponível em: https://pt.wikipedia.org/wiki/Recombina%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o_evolutiva). Acesso em: 17 mai. 2019.

RYU. **Ryu SDN Framework**. 2018. Disponível em: https://ryu-sdn.org/. Acesso em: 06 fev. 2022.

WANG, W.; SUN, Y.; ZHENG, K.; KAAFAR, M.; LI, D.; LI, Z. Freeway: Adaptively isolating the elephant and mice flows on different transmission paths. In: **2014 IEEE 22nd International Conference on Network Protocols (ICNP)**. Los Alamitos, CA, USA: IEEE Computer Society, 2014. p. 362–367. Disponível em: https://doi.ieeecomputersociety.org/10.1109/ICNP.2014.59.

ZAHAVI, E.; KESLASSY, I.; KOLODNY, A. Quasi fat trees for hpc clouds and their fault-resilient closed-form routing. In: **2014 IEEE 22nd Annual Symposium on High-Performance Interconnects**. [*S. l.*: *s. n.*], 2014. p. 41–48.

ZHANG, Y.; CUI, L.; CHU, Q. Fincher: Elephant flow scheduling based on stable matching in data center networks. In: **2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)**. Nanjing, China: IEEE, 2015. p. 1–2.