UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Davison Holanda Pacheco

Q-PATH: UM ALGORITMO DE CÁLCULO DE CAMINHOS COM PARÂMETROS DE QUALIDADE DE SERVIÇO PARA REDES DEFINIDAS POR SOFTWARE.

Belém
2016

Davison Holanda Pacheco

Q-PATH: UM ALGORITMO DE CÁLCULO DE CAMINHOS COM PARÂMETROS DE QUALIDADE DE SERVIÇO PARA REDES DEFINIDAS POR SOFTWARE.

Dissertação de Mestrado apresentada como requisito para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas e Naturais da Universidade Federal do Pará. Área de concentração: Redes de Computadores. Linha de pesquisa: Redes definidas por software.
Orientador: Prof. Dr. Antônio Abelém.

Belém

2016

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DAVISON HOLANDA PACHECO

Q-PATH: UM ALGORITMO DE CÁLCULO DE CAMINHOS COM PARÂMETROS DE QUALIDADE DE SERVIÇO EM REDES DEFINIDAS POR SOFTWARE

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Pará como requisito para obtenção do título de Mestre em Ciência da Computação, defendida e aprovada em 28/12/2016, pela banca examinadora constituída pelos seguintes membros:

Prof. Dr. Antônio Jorge Gomes Abelém
Orientador – PPGCC/UFPA

Prof. Dr. Eduardo Coelho Cerqueira
Membro Interno – PPGCC/UFPA

Prof. Dr. Raimundo Viegas Junior
Membro Externo – FACOMP/UFPA

Visto:
Prof. Dr. Jefferson Magalhães de Morais
Vice-Coordenador do PPGCC/UFPA

Prof. Dr. Jefferson Magalhães de Morais
Coordenador do PPGCC

Davison Holanda Pacheco

Q-PATH: UM ALGORITMO DE CÁLCULO DE CAMINHOS COM PARÂMETROS DE QUALIDADE DE SERVIÇO PARA REDES DEFINIDAS POR SOFTWARE.

Dissertação de Mestrado apresentada como requisito para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas e Naturais da Universidade Federal do Pará.

Data de aprovação: ____ / ____ / 2016, Belém-PA.

Banca Examinadora

Prof. Dr. Antônio Jorge Gomes Abelém

PPGCC – UFPA – Orientador

Prof. Dr. Eduardo Coelho Cerqueira.

PPGCC – UFPA – Membro Interno

Prof. Dr. Raimundo Viégas Junior

FACOMP – UFPA – Membro Externo

Belém

2016

A meu Deus, pais e esposa, pelo amor incondicional.

Aos meus irmãos e amigos que apoiam em todos os momentos.

“Fé é a certeza de coisas que se esperam e a convicção de fatos que não se veem”; “*Tudo posso naquele que me fortalece*”.

Bíblia Sagrada (Hebreus 11:1; Filipenses 4:13)

AGRADECIMENTOS

Para quem acredita na existência de um ser superior, é comum iniciar um agradecimento fazendo referência a **Deus**. Por isso e por ser um cristão, ter plena convicção da existência desse ser e a certeza que olha pra cada ser humano, gostaria de fazer essa referência a Ele. Entre tantos acontecimentos que me motivaram a desistir, uma força maior sempre me direcionava a seguir em frente e nunca desistir. Acredito que foram, também, as orações de meus pais todos os dias, o cuidado e carinho que **Deus** tem por mim. De uma coisa tenho certeza, se consegui concluir é porque Ele me conduziu em cada momento. Por isso Lhe sou imensamente grato, por se importar comigo e conduzir minha vida.

Agradeço a meus pais, **Natanael Santos Pacheco** e **Espedita Holanda Pacheco**, mesmo sem ter conhecimento da dimensão que é fazer um mestrado, eles sempre acreditaram em mim e me motivaram a ir em frente. Eles são um dos maiores motivos para eu ter tido tanta força pra continuar, possivelmente este também seja parte do sonho deles pra minha vida, e mesmo que estejam vivendo em uma roça todos os dias, estão felizes/satisfeitos de ver um filho conseguindo superar barreiras e conquistando títulos considerados até "impossíveis" para aquele meio. Meus pais sempre são meus guerreiros e motivadores de mudança.

À minha amada esposa, **Mayara del Aguilal Pacheco**, pelas tantas vezes que conversamos. Durante a eminência de dar um novo rumo à minha vida, ela sempre me dava força pra superar as dificuldades e estava disposta a me apoiar no que precisasse, porém sempre aconselhando que eu finalizasse o mestrado. Algumas vezes cheguei a ficar um pouco desorientado quanto ao que seguir e o que fazer, mas ela sempre ajudava, acalmando-me, pedindo para pedir orientação Divina. Foi com o apoio dela e de Deus que, quando tudo começou, aparentemente, a dar errado (quando queimou primeiro meu HD externo, depois meu computador do trabalho onde eu tinha feito backup; em seguida minha tv, juntamente com o monitor do meu PC; e, por fim, na escrita dos agradecimentos, quando tive que pedir emprestado à vizinha o net dela porque queimou a tela de meu notebook), tive forças e uma certeza interior que eu estava no caminho certo. A cada acontecimento, era em minha esposa que eu encontrava mais força pra concluir. Foi com o apoio dela que superei cada dificuldade.

Minha imensa gratidão a meu amigo **Airton Ishimori,** que foi mais que um coorientador, um grande amigo, por momentos teve que me aturar durante horas no telefone quando eu tinha dúvidas; além da paciência de revisar várias vezes o trabalho, sempre com muitas críticas construtivas (apesar de ser doloroso retirar textos que algumas vezes eu tinha demorado dias escrevendo ☺) e, principalmente, por ter me aceitado como orientando

faltando tão pouco tempo pra concluir. Seu apoio foi, também, o que me fez conseguir escrever este trabalho. Agradeço de forma especial também à minha amiga **Fabiola Gomes**, que acredito entender do meu trabalho tanto quanto eu, de tantas e tantas vezes que ela, pacientemente, aceitou revisar, sempre com muito bom humor em suas revisões. A meu amigo **Matakura** (Leonardo Sarraff), que sempre me motivou, enfrentando as mesmas dificuldades pra concluir o mestrado, sempre me ajudando quando precisava de inglês, programação, enfim, sempre disponível.

Agradeço também a meus amigos e irmãos que direta ou indiretamente me apoiaram nessa jornada de alegrias e tristezas. Dentre muitos gostaria de citar, **Aline Gomes**, **Valber Pacheco**, **Suzane Pacheco**, **Hélia Pacheco**, **Elda Pacheco**, **Raphael Paiva**, **Joelyson Rodrigues**, **Fernando Farias**, **Mario Ribeiro**, **Sandro Monteiro**, **Danilo Souza**, **Abraão Pinto** e **Danilo Rosa**. Cada um teve sua contribuição específica e de grande valor, infelizmente, o espaço é pequeno pra escrever, mas os agradeço muito. Aos amigos que não citei, mas que também me apoiaram, quero dizer que sou muito, muito grato a cada um.

É costume agradecer ao professor, possivelmente já se espera por isso, mas queria frisar que realmente sou imensamente grato ao professor **Antônio Abelém;** se consegui finalizar e vencer essa etapa de minha vida é porque tive seu apoio, depois de ter me afastado tantas fezes, ainda assim, permitiu que eu retornasse para concluir. Sou muito grato pelas cobranças, pelo apoio com o laboratório e, principalmente, por ter acreditado em mim no dia que fiz a entrevista pra entrar no mestrado. Atos simples muitas vezes fazem toda a diferença na vida de uma pessoa, por isso meu muito obrigado mais uma vez.

Não poderia deixar de agradecer à banca composta pelo professor **Eduardo Cerqueira** e **Raimundo Viégas,** os quais prontamente aceitaram participar deste momento tão importante para mim. Professores que admiro pelo talento, conhecimento, além de muita simplicidade.

Por fim, sou grato ao programa PPGCC da UFPA, juntamente o corpo docente de excelentes professores, representados pela pessoa do professor **Abelém**. Além destes agradeço também à Capes pela oportunidade concedida a mim para a conquista deste título.

RESUMO

O paradigma de Redes Definidas por Software (SDN) vem sendo investigado como a solução mais promissora para o engessamento da Internet, uma vez que propõe a dissociação entre o plano de dados e o plano de controle, proporcionando maior programabilidade às redes. No entanto, há lacunas em serviços disponíveis nesse modelo, dentre as quais se observa o serviço de encaminhamento dos fluxos, baseados em critérios de qualidade de serviço (QoS). Por exemplo, a reserva de recursos, a partir dos requisitos necessários de cada aplicação, permanece como um desafio a ser vencido. Nesse contexto, o presente trabalho apresenta uma proposta de algoritmo de cálculo de caminhos para SDN chamada Q-Path. O algoritmo busca o melhor caminho, baseando-se em parâmetros de QoS da rede, possibilitando restrições determinísticas impostas pelas aplicações de usuário ou por parte do administrador da rede. A proposta fundamenta-se na utilização dos parâmetros de QoS de perda de pacotes e a latência, como condicionadores para a busca do melhor caminho. A análise dos resultados obtidos com extensas simulações demonstram que a proposta é capaz de reduzir o tempo de busca do caminho em aproximadamente 65%, quando comparado com o algoritmo Dijkstra, comumente utilizado como o algoritmo motriz de serviço de encaminhamento dos pacotes. Além disso, a proposta consegue melhorar o consumo de memória e processamento em torno de, respectivamente, 11% e 9%, durante a busca do melhor caminho.

PALAVRAS-CHAVE: Algoritmo. *OpenFlow*. Qualidade de Serviço. Redes Definidas por Software.

ABSTRACT

The Software Defined Networks (SDN) paradigm has been investigated as the most promising solution for the inflexibility of the Internet, since it proposes the dissociation of the data plan and control plan, providing greater programmability to the networks. However, this model presents some gaps like the flow routing service, based on quality of service (QoS) criteria. For example, reserving resources, from the required requirements of each application, remains a challenge to be overcome. In this context, this work presents a proposed algorithm for calculating paths for SDN called Q-Path. The algorithm searches for the best path based on network QoS parameters allowing deterministic constraints imposed by the user's applications or by the network administrator. The proposal is based on the use of QoS parameters of packet loss and latency, as conditioners to search for the best path. The analysis of the results obtained with extensive simulations demonstrates the proposal is able to reduce the search time of the path by approximately 65% when compared to the Djikstra algorithm, commonly used as the main algorithm of the packet forwarding service. In addition, the proposal improves memory consumption and processing time, respectively, around 11% and 9% while searching for the best path.

KEYWORDS: Algorithm. *OpenFlow*. Quality of Service. Software Defined Networking.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

Anatel	Agência Nacional De Telecomunicações
BFS	Breadth-First Search
BYOA	Bring Your Own App
BYOD	Bring Your Own Device
DFS	Depth-First Search
DiffServ	Differentiated Service
Gbps	Gigabits Por Segundo
IntServ	Integrated Service
IP	Internet Protocol
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MILPFlow	Mixed Integer Linear Programming with OpenFlow
MPTCP	MultiPath TCP
ODL	Opendaylight
ONOS	Open Network Operating System
OVS	Open Virtual Switch
PMT	Período de Maior Tráfego
PoP	Point of Presence
QoS	Quality of Service
REST	Representational State Transfer
RNP	Rede Nacional De Pesquisa
RSVP	Resource Reservation Protocol
SDN	Software Defined Networking
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol

Tbps	Terabits por Segundo
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo introdutório é apresentado o contexto geral deste trabalho. Além de uma visão geral sobre a área a qual o trabalho está inserido, são abordados os desafios que justificam a escolha do tema, objetivos a serem alcançados, bem como a estrutura elaborada para a apresentação.

1.1 Visão Geral

Concomitantemente ao aumento do desenvolvimento tecnológico e acessibilidade à Internet, surgiram inúmeras aplicações com interações entre os usuários, além de projetos científicos que necessitam de muita interação entre os colaboradores. Paralelamente, a comunidade científica tem criado projetos de grande magnitude, de tal forma que o sucesso dos projetos necessite de muita colaboração. Dentre tais projetos, convém destacar o LHC (Large Hadron Collider) (Millington et al., 2016) que é considerado o principal acelerador de partículas da comunidade científica atualmente, necessitando de alto processamento e compartilhamento de informação.

Sobre as aplicações de usuários com interações online de tempo real, middleton; Modafferi (2016) destacam algumas que necessitam de requisitos específicos de latência e perda de pacotes para que tenham um bom desempenho, tais como Minecraft, Quake 3 Urban Terror, etc. Assim, vem se tornando comum o uso de teleconferências, aulas à distância em tempo real e serviços de *vídeo streaming*.

Entretanto, garantir qualidade de serviço (*Quality of Service* - QoS) para os usuários finais é um desafio que vem se agravando nas últimas décadas. Muitos pesquisadores, como Moraes (2015) e guedes et al. (2012), afirmam que a arquitetura TCP/IP (*Transmission Control Protocol/ Internet Protocol*), padrão da Internet, atingiu um nível de amadurecimento que a torna pouco flexível, transformando-se em um grande desafio para as gerências de serviços. O resultado dessa arquitetura é comumente denominado pela comunidade científica como "engessamento da Internet" (Chowdhury; Boutaba, 2009).

Segundo Kurose (2013), arquitetura de rede refere-se à organização do processo de comunicação em camadas (por exemplo, as cinco camadas da arquitetura da Internet).

Mesmo com a criação de alguns projetos como as arquiteturas *Integrated Service / Resource Reservation Protocol* (IntServ/RSVP) (Bradenand et al., 1997), *Differentiated Service* (DiffServ)

(Kasigwa; Baryamureeba; Williams, 2007) e *Classless Inter-Domain Routing* (CIDR) (Fuller et al., 1993), mantém-se a complexidade ao oferecer QoS na arquitetura TCP/IP.

Em contraste com a arquitetura TCP/IP, novos paradigmas de redes de computadores surgiram objetivando possíveis direcionamentos para o futuro das redes de computadores. Dentre esses paradigmas, encontram-se as Redes Definidas por Software (SDN - *Software Defined Networking*), modelo idealizado por Casado et al. (2009). Kobayashi et al. (2014) explanam sobre o modelo SDN, afirmando que há uma ruptura ou divisão dos planos de atuação (plano de controle e plano de dados) e, dessa maneira, a inteligência da rede (plano de controle) passa a ser logicamente centralizada, utilizando *softwares* controladores, enquanto os dispositivos de encaminhamento (plano de dados) se tornam programáveis por meio de um protocolo de controle aberto. Em outras palavras, este protocolo possibilita a comunicação padronizada entre os planos de controle e dados. Atualmente, na literatura, OpenFlow (Mckeown et al., 2008) é o protocolo mais conhecido de SDN e implementado pela comunidade acadêmica e pela indústria.

O modelo SDN oferece a possibilidade de programar o plano de controle de tal modo que reflita em ações aplicadas ao plano de dados, utilizando interfaces abertas. Nesse contexto, é possível efetuar um gerenciamento na rede, de forma dinâmica, sem muitas complexidades (envolvendo hardware e software proprietários). Contudo, realizar a computação de caminhos e garantir um nível de QoS satisfatório em SDN para os usuários da rede são dois tópicos importantes que devem ser considerados.

Com o intuito de oferecer QoS ao usuário, vários autores propuseram técnicas de como implementá-lo, focando em necessidades exigidas por algumas aplicações de usuários, como citado anteriormente, que precisam de configurações específicas de rede para um bom desempenho. Nos trabalhos apresentados por Nobre (2014), Brito et al., (2014) e Tomovic; Prasad; Radusinovic (2014), por exemplo, há a proposição de criações de circuitos, efetuando buscas e configurações de caminhos por meio de algoritmos já consagrados na literatura (por exemplo, Dijkstra).

Devido a todo o conhecimento da rede em SDN, é possível oferecer caminhos conforme as necessidades do usuário, seja com uma largura de banda específica, seja com uma latência mínima, ou até mesmo que não se passe por filas com alto índice de bloqueio e etc. Entretanto, o fator desempenho deve ser considerado como o principal fator nessas soluções, independente de métrica de QoS, pois garantir um nível de QoS e efetuar a computação de caminhos são dois tópicos opostos em quesito de desempenho. Por um lado, o excesso de parâmetros de rede podem tornar a computação de rotas complexas e custosa, em termos de processamento, ao considerar

relacionamento entre múltiplos parâmetros de QoS. Isso pode garantir um nível de QoS satisfatório, mas o desempenho da rede pode ser prejudicado devido ao custo de processamento excessivo no plano de controle. Por outro lado, um número limitado de parâmetros de QoS na computação de rotas pode garantir um tempo de resposta mais eficiente, porém, com níveis de QoS insatisfatórios. Portanto, é necessário que o algoritmo de cálculo de caminhos apresente um custo de processamento mínimo e garanta um nível de QoS satisfatório.

A literatura revela que, geralmente, os pesquisadores, como os já citados, utilizam apenas um desses requisitos de QoS, ou, se combinadas a métricas, não se preocupam com o tempo de resposta ao se efetuar a busca. Isso pode exigir um processamento maior e um tempo de busca mais demorado. Em síntese, nos trabalhos científicos encontrados, escolhe-se uma das métricas de QoS e se aplica ao algoritmo de Dijkstra.

1.2 Motivação e Justificativas

Alguns trabalhos, como Yilmaz; Tekalp; Unluturk (2015) e Bhattacharya; Das (2013), apresentam novas soluções, utilizando SDN, para suprir as necessidades de QoS demandadas pelas aplicações emergentes. Entretanto, permanecem arraigadas aos métodos antigos. Por exemplo, utilizam algoritmo de Dijkstra ou outros da literatura sem aproveitar o potencial de informações disponíveis no controlador, utilizando apenas um parâmetro de busca para o algoritmo. As aplicações de usuários emergentes, em geral, exigem QoS específicos para um bom funcionamento; e o controlador SDN possui o conhecimento das informações de rede. Assim, torna-se desperdício não aproveitar o potencial ofertado por SDN.

Em Tomovic; Radusinovic; Prasad (2015), encontra-se uma proposta de *framework* para prover QoS para aplicações multimídia de forma automática e flexível. No entanto, apesar de impor algumas condições na oferta de QoS, ao final recai sobre o algoritmo de Dijkstra original, para encontrar o melhor caminho utilizando a largura de banda da rede como peso para as arestas.

Apesar do algoritmo de Dijkstra ser consagrado na literatura, devido sua maior eficiência ao buscar um caminho, com um vértice de origem definido, é possível incluir modificações no algoritmo, melhorando seu desempenho na busca de caminhos, aproveitando as vantagens de SDN, cuja otimização faz parte dos objetivos da presente dissertação.

A Norma G.114 do ITU-T (*International Telecommunication Union - Telecommunication Standardization Sector*), esclarece que atrasos totais no sistema entre 0 e 150 ms são aceitáveis para a maioria das aplicações; entre 150 e 400ms deve ser avaliado o impacto na qualidade da

aplicação (sendo os casos típicos onde é usado satélite como parte da solução de transmissão); e superior a 400ms, geralmente, é inaceitável, salvo os casos nos quais não existe outra forma de acesso senão com duplo salto de satélite ou quando a aplicação não exija forte interação entre os usuários. Denota-se que, assim como a delimitação apresentada em Gorlatch; Humernbrum; Glinka (2014), nas aplicações de tempo real, outras aplicações de usuários também podem ter outras características específicas.

Adicionalmente, no Brasil, a Anatel (Agência Nacional de Telecomunicações) lançou a Resolução Nº 574[1] (publicada em outubro de 2011 e atualizada em agosto 2016) que trata sobre a importância da latência, bem como da perda de pacotes, afirmando, respectivamente, que: "Art. 18. Durante o PMT (Período de Maior Tráfego), a Prestadora deve garantir latência bidirecional de até oitenta milissegundos (terrestre) e novecentos milissegundos (satélite)" e "Art. 20. Durante o PMT, a Prestadora deve garantir que a percentagem de pacotes descartados seja de até dois por cento".

A resolução nº 574 é mais uma demonstração do quanto é importante lidar com os parâmetros de latência e perda de pacotes abordados nesse trabalho.

A oportunidade trazida por SDN para melhorar buscas usando combinações de requisitos de QoS motiva o desenvolvimento do presente estudo, pois com informações da rede é possível inserir condições dentro da busca, possibilitando encontrar o caminho almejado de forma mais rápida. Portanto, pretende-se reduzir a carga de processamento do controlador e agilizar o tempo de resposta das solicitações de caminhos, auxiliando o controlador para que este não se torne um gargalo.

Nesse contexto, o presente trabalho pretende contribuir com a área de pesquisa e desenvolvimento de SDN por meio da apresentação do algoritmo QoS Path (Q-Path), objetivando que soluções inovadoras integradas a esta proposta agreguem benefícios aos cenários de experimentação requeridos pela área de Internet do Futuro, sugerindo novas soluções para o mercado. O Q-Path é uma solução de busca de caminhos baseado no algoritmo de Dijkstra que utiliza o conhecimento globalizado dos recursos da rede para efetuar buscas de caminhos, a partir de requisitos de QoS exigidos pelas aplicações de usuário ou por um SLA (*Service Level Agreement* - Acordo de nível de serviço). Por exemplo, podem ser citadas aplicações em tempo real que, segundo Gorlatch; Humernbrum; Glinka (2014), necessitam de um tempo mínimo de

[1] http://www.anatel.gov.br/legislacao/resolucoes/26-2011/57-resolucao-574

resposta ideal de 100 ms. Com necessidades como essa, por parte das aplicações, é possível usar o algoritmo para encontrar um caminho que possua uma distância limitada (por exemplo, 100 ms), não sendo necessário verificar roteadores que estejam fora do alcance desse limite.

1.3 Contribuição do trabalho

Diante do exposto, o presente estudo baseou-se na apresentação de uma solução de algoritmo baseado no Dijkstra. A solução aproveita o conhecimento da rede disponível no controlador para efetuar uma busca, utilizando uma combinação de dois requisitos de QoS (perda de pacotes e latência), objetivando oferecer um caminho com mais qualidade, diminuir o tempo de busca dos caminhos, além de reduzir a carga de processamento no controlador. As avaliações da proposta, no presente trabalho, apresentam um estudo comparativo em que a proposta atinge resultados positivos no consumo de processamento e no aprovisionamento de QoS.

1.4 Objetivos

1.4.1 Objetivo geral

Desenvolver um algoritmo que encontre o caminho com menor latência dentro das condições impostas por uma aplicação ou administrador de rede, mediante parâmetros de QoS para redes definidas por software. Com isso pretende-se melhorar o tempo de busca do caminho, bem como reduzir o custo de processamento e memória no controlador.

1.4.2 Objetivos específicos

Como desdobramento dos objetivos gerais, são listados abaixo os seguintes objetivos específicos:

- Avaliar o escopo de mensagens OpenFlow e suas possíveis aplicabilidades no que tange ao controle dos dispositivos de rede;
- Criar métodos que identifiquem cada alteração na rede para efetuar a descoberta de recursos e dados estatísticos na rede;
- Realizar estudo e levantamento das mensagens de status fornecidas pelo protocolo OpenFlow (mensagens *OpenFlowStats*);
- Desenvolver uma estrutura de grafo para armazenar a topologia da rede detectada a partir

das mensagens Openflow.

- Implementar o algoritmo pra descoberta de caminhos;
- Desenvolver um módulo de integração com o núcleo do controlador para que esse possa utilizar o algoritmo;
- Definir os parâmetros de rede pertencentes ao escopo deste trabalho para a busca do caminho;
- Desenvolver interfaces para consumo de serviços por meio de chamadas de serviços web;
- Descrever os resultados dos experimentos concernentes ao atendimento das necessidades funcionais do algoritmo e aspectos de performance;
- Comparar os resultados obtidos nos experimentos com soluções similares.

1.5 Estrutura da Dissertação

Além do capítulo introdutório, o trabalho é composto por mais cinco capítulos divididos seguindo o ordenamento descrito abaixo:

Capítulo 2: Conceitos de SDN, protocolo OpenFlow, aplicações de controladores OpenFlow e algoritmos de busca.

Capítulo 3: Propostas de trabalhos relacionadas aos objetivos deste trabalho.

Capítulo 4: Proposta do algoritmo de busca, descrevendo de forma detalhada o funcionamento do algoritmo e apresentando uma arquitetura com os módulos que integram o algoritmo ao núcleo do controlador SDN.

Capítulo 5: Os cenários de experimentação para validação da proposta, assim como características de desempenho que validam a aplicabilidade do trabalho.

Capítulo 6: As considerações finais sobre a pesquisa, além de propostas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão descritos os principais conceitos para o desenvolvimento deste trabalho. Inicialmente, são apresentados os conceitos relativos à SDN, mostrando como ocorreu a expansão desse paradigma, bem como aspectos relativos à sua operação. Em seguida, são apresentadas as características e o funcionamento dos controladores SDN, além do detalhamento de protocolo OpenFlow.

2.1 Limitações da Arquitetura de Rede Tradicional

Casado et al. (2009) afirmam que as redes de computadores se tornaram um ponto crítico em praticamente todas as áreas de negócio. Existem milhares de aplicações que funcionam sobre uma estrutura de *switches*, roteadores e *bridges*, que necessitam de características como: confiabilidade, disponibilidade, integridade, baixa latência, flexibilidade e escalabilidade. Entretanto, apesar do aumento das capacidades dos canais de comunicação e maior poder de processamento dos equipamentos de rede, a rede em sua infraestrutura, do ponto de vista arquitetural, apresentou poucas mudanças. Fabricantes criam equipamentos como uma caixa-preta, de modo que o *firmware* esteja encapsulado e fechado no *hardware*, fundindo-os como se fossem uma entidade única. Assim, empresas constroem suas redes com um aglomerado de caixas-pretas interligadas, levando o mercado a uma busca por desenvolver melhores equipamentos, atrasando o desenvolvimento da arquitetura TCP/IP.

Essa arquitetura estabelecida é frequentemente referenciada, do ponto de vista estrutural, como algo bom e ao mesmo tempo ruim. A forma como a evolução da rede ocorreu, criou como base um emaranhado de protocolos e camadas de equipamentos que, apesar de permitir a comunicação da forma como conhecemos hoje, ergueu uma enorme barreira para inovações e experimentações necessárias à área de redes de computadores (Spalla, 2015). Segundo Mckeown et al. (2008), pouca tecnologia é transferida da academia para o mercado, e, geralmente, a partir de sua concepção, uma inovação leva até 10 anos para entrar em produção.

Tradicionalmente, essa arquitetura impõe diversas barreiras para os administradores de redes. Por exemplo, se o administrador de redes deseja realizar alguma alteração na lógica de tratamento de pacotes (roteamento/encaminhamento), muitas das vezes o fabricante não possibilita ao pesquisador desenvolver novos algoritmos de tratamentos de pacotes para serem instaladas e configuradas para execução no sistema embarcado do *hardware*, implantado pelo fabricante, pois

apenas uma interface de alto nível é disponibilizada para configuração do *hardware*. Consequentemente, a adição de novas características significa novos produtos ou novas licenças de software, o que cria uma profunda dependência dos fabricantes de *hardware*.

A Figura 1.a apresenta um exemplo de equipamentos com arquitetura fechada. Nela destaca-se que em um mesmo equipamento encontram-se uma parte para o encaminhamento de pacotes e uma parte inteligente efetuando o controle dos fluxos, utilizando protocolos específicos de encaminhamento e aplicações ou ferramentas específicas do fabricante.

Alguns pontos que motivaram a comunidade científica em propor um novo paradigma de redes de computadores são:

I. Novos padrões de tráfego: os grandes centros de dados de empresas que provêm serviços de computação em nuvem vêm mudando os padrões de tráfego da Internet. Ao contrário de uma comunicação cliente-servidor tradicional, na qual um cliente se conecta a um servidor em uma comunicação "vertical", hoje, uma simples requisição de pesquisa na Internet exige uma comunicação sincronizada entre diversos servidores e banco de dados distribuídos geograficamente separados, significando que, para a requisição, esse fluxo de dados irá envolver diversos pontos da rede antes de retornar ao usuário. Além disso, usuários se conectam e interagem com a rede com uma maior dinâmica nos dias atuais, via dispositivos móveis (ex.: *smartphones* e *tablets*) e aplicações e-commerce. Cada dispositivo ou aplicações de usuário tem sua forma de consumir e gerar dados na Internet, o que torna o tráfego na rede mais dinâmico.

II. Consumerização de TI: usuários cada vez mais fazem acesso às redes corporativas a partir de diferentes dispositivos como *smartphones* e *tablets*. Desse modo, existe uma necessidade crescente de configurações específicas de acordo com o perfil dos usuários, a fim de permitir o acesso à rede, ao mesmo tempo em que os níveis de segurança permaneçam. Essa mudança, na qual *bring your own device* (BYOD) e *bring your own app* (BYOA) (Earley et al., 2014) foram os principais agentes transformadores, trouxe um grande desafio para a gerência de TI, já que são comportamentos que estão em plena evolução e condizem com a perspectiva atual de TI, em que a tecnologia está cada vez mais presente, intuitivamente.

III. Computação na nuvem e virtualização: com o advento da computação na nuvem, empresas vislumbraram diversas possibilidades para seus negócios, o que resultou em um crescimento expressivo das soluções em nuvem. Todavia, manter uma infraestrutura do tipo *cloud* envolve requisitos de segurança, disponibilidade, escalabilidade, auditoria, mudanças das regras de negócio e grande volume de dados, o que significa uma estrutura complexa. Entretanto, ainda assim, é desejável que todo o gerenciamento da infraestrutura seja simples.

2.2 Paradigma de Redes Definidas por Software

As SDNs surgiram como um promissor conceito para o projeto de novas arquiteturas para a Internet. O ponto central desse modelo e a separação física dos planos de controle (responsável pelos protocolos e pelas tomadas de decisão que resultam na criação e atualização das tabelas de encaminhamento) e de dados (comumente chamado de plano de encaminhamento, que controla a comutação e o repasse dos pacotes de rede) com uma comunicação independente de fornecedor, para o mecanismo de encaminhamento, como OpenFlow (Mckeown et al., 2008).

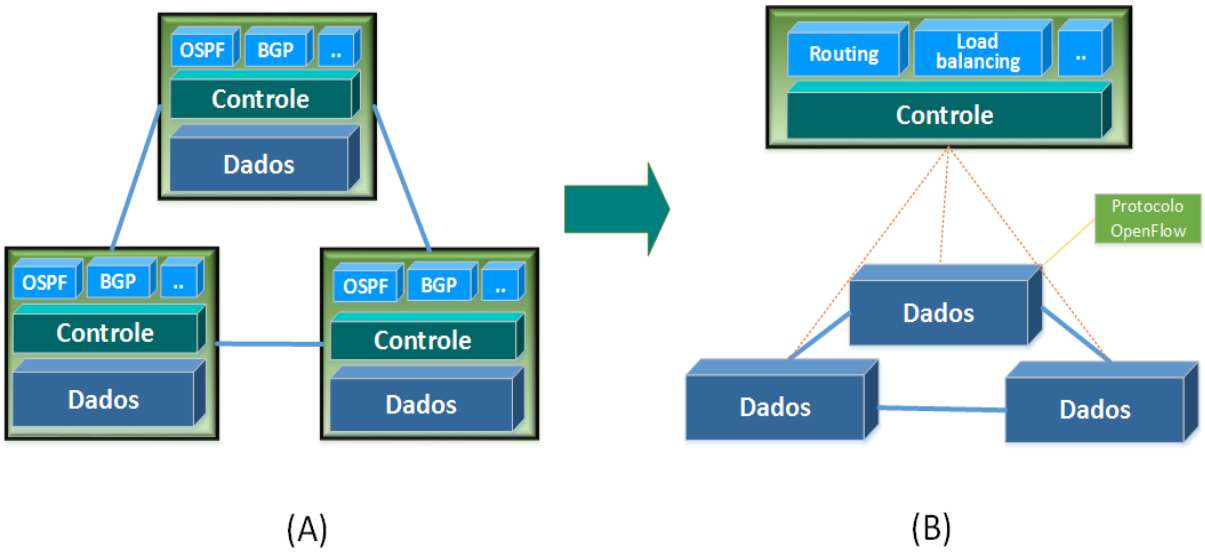
A principal consequência é que as funcionalidades da rede podem ser facilmente alteradas ou mesmo definidas, após a rede ter sido implantada. Novas funcionalidades podem ser adicionadas, sem a necessidade de se modificar o hardware, permitindo que o comportamento da rede evolua na mesma velocidade que o software.

Convém destacar que em SDN, entre os diversos aspectos inovadores, o mais importante é o alto grau de flexibilidade oferecido pela programabilidade da rede, uma vez que os projetistas de redes têm a possibilidade de configurar e escrever suas aplicações por meio de uma camada de abstração de alto nível, em vez de aguardar um *update* de *firmware* ou um novo produto do fabricante. Isso permite que a cada alteração na lógica de encaminhamento da rede ou adição de uma nova *feature*, o código da aplicação seja simplesmente atualizado. Como a camada de aplicação independe do hardware, a evolução da aplicação pode ser dinâmica e constante. Assim, toda a capacidade oferecida por uma linguagem de programação está disponível para utilização em uma aplicação SDN.

A Figura 1 compara as arquiteturas de redes tradicionais com SDN. Enquanto as redes tradicionais (Figura 1.a), como TCP/IP, não separam o plano de controle e o plano de dados, em SDN (Figura 1.b), o plano de controle é retirado de cada *switch* e disponibilizado em um único ponto externo, passando a controlar a rede remotamente.

Objetivando uma ruptura entre a parte inteligente da rede e a parte de encaminhamento de pacotes, na arquitetura SDN, toda a inteligência operativa da rede encontra-se centralizado em um único ponto, ou seja, permite uma visão centralizada. Esse modelo traz consigo um grande avanço para a área de redes, pois possibilita aos utilizadores a definição de fluxos de dados e determinação dos caminhos dos fluxos usando software, independentemente do *hardware*, o que antes era inviável. O modelo SDN traz um conceito de redes abertas, no sentido de tornar-se possível integrar *software* e *hardwares* de diferentes fabricantes.

Figura 1 – Apresenta um paralelo entre a arquitetura de rede TCP/IP e o modelo SDN.



Fonte: Adaptado de (Carlos, 2016).

Outro ponto marcante deste modelo é que não há mais a necessidade de ter réplicas de protocolos e aplicações específicas de encaminhamento de pacotes, pois tudo está centrado no controlador, onde se encontra a inteligência da rede. Desta forma, a introdução de aplicações e funcionalidades torna-se uma questão de instalação de pequenos programas, elaborados por quem os quiser programar.

O modelo SDN é fundamentado em três camadas. Uma camada de aplicações, na qual constam as aplicações comerciais que rodarão sobre o modelo SDN, como jogos, videoconferência, streaming, etc. Uma camada de controle dos serviços de rede, seja de encaminhamento, seja busca de rotas, etc. E, por fim, uma camada de dados com os respectivos *switches* de encaminhamento de pacotes, essas duas últimas camadas são apresentadas na Figura 1.b

Como elo entre o controlador de as demais extremidades, pode-se destacar o OpenFlow como o protocolo mais utilizado para interligar o controlador à camada *southbound;* para interligar à ponte *northbound,* utilizam-se as linguagens de interoperabilidade com seus respectivos protocolos, tais como REST/JSON (*REpresentational State Transfer / JavaScript Object Notation*), SOAP/XML (*Simple Object Access Protocol / eXtensible Markup Language*), etc.

A seguir destacam-se algumas características inerentes ao uso de SDN's (FOUNDATION,):

- Gerenciamento e controle centralizado dos equipamentos de múltiplos fabricantes;

- Maior capacidade de inovação, já que novas ferramentas e serviços podem surgir por meio de aplicações desenvolvidas, ou seja, por desenvolvimento de software, sem a necessidade fazer alterações no *hardware*;
- Controle granular da rede, com possibilidade de aplicar diferentes políticas de segurança a diferentes perfis de usuários e equipamentos;
- Melhoria da automação do gerenciamento, usando diferentes APIs (*Application Programming Interface*) para abstrair os detalhes da rede para outras aplicações, como sistemas de orquestração, sistemas de aprovisionamento e aplicações de rede.

Ainda na Figura 1.b nota-se a estruturação do modelo SDN através do protocolo OpenFlow, no qual o plano de controle consiste em um controlador OpenFlow, e o plano de dados consiste nos *switches* OpenFlow, sendo que esses dispositivos são conectados ao controlador por meio do protocolo de controle OpenFlow.

O *switch* pode ser físico ou virtual e possui três componentes distintos: tabela de fluxos, canal seguro e o protocolo de comunicação com o controlador exemplificado pelo protocolo OpenFlow. Abaixo são descritos tais componentes.

1. Tabela de Fluxos: A entrada na tabela de fluxos de uma SDN consiste em regras, ações e contadores. Cada fluxo é associado a uma ação, e, assim, os pacotes que chegam ao equipamento são encaminhados de acordo com as regras instaladas na tabela de fluxos. Os contadores são usados para manter estatísticas de utilização, além de servirem para remover fluxos inativos.
2. Canal Seguro: o canal conecta o equipamento ao controlador remoto, utilizando uma conexão criptografada, permitindo que mensagens e pacotes sejam enviadas entre o controlador e o *switch*; isso é necessário para evitar ataque de elementos mal-intencionados, haja vista que a rede é desenvolvida com base em protocolos públicos, ou seja, abertos, *open source*;
3. O protocolo OpenFlow: o protocolo define um padrão aberto de comunicação entre o controlador e o *switch*, que permite a programação da tabela de fluxos do equipamento por meio de uma interface de alto nível.
4. Controlador: É o software responsável por tomar decisões e adicionar e/ou remover as entradas na tabela de fluxos. A programação do controlador permite a evolução das tecnologias nos planos de dados e as inovações na lógica das aplicações de controle.

2.3 Protocolo Openflow

O protocolo OpenFlow foi criado como uma solução para utilização em *switches Ethernet* para a construção de uma rede personalizada para experiências acadêmicas (Mckeown et al., 2008). Evitando a grande quantidade de protocolos existentes, ele foi desenvolvido objetivando flexibilidade e possibilitando uma implementação de alto desempenho e baixo custo.

Desde sua origem, em Maio de 2008, com a versão 0.2.0, o protocolo está em constante evolução, chegando, atualmente, à versão 1.5, lançada em dezembro de 2014. Dentre os lançamentos, tem-se a versão 1.0.0 lançada em Dezembro de 2009; posteriormente, foram lançadas as versões 1.1, 1.2, 1.3, e 1.4 em fevereiro de 2011, dezembro de 2011, junho de 2012 e outubro de 2013, respectivamente.

O protocolo é um padrão em desenvolvimento para administração de redes LAN e WAN, possibilitando assim o controle e a criação de VLANs, roteamentos e qualidade de serviço, oferecendo uma padronização e permitindo que diversos fabricantes usem um conjunto de regras padronizadas, visando a inclusão de novas características e protocolos, independentes do software utilizado no *switch* (Duque et al., 2012).

Para criação de um fluxo utilizando o protocolo OpenFlow é necessário um *switch* que o suporte, um software externo para controlar o fluxo e definir as regras cabíveis ao fluxo (representado pelo controlador) e um canal seguro para comunicação entre os dois componentes.

2.4 Plano de Controle

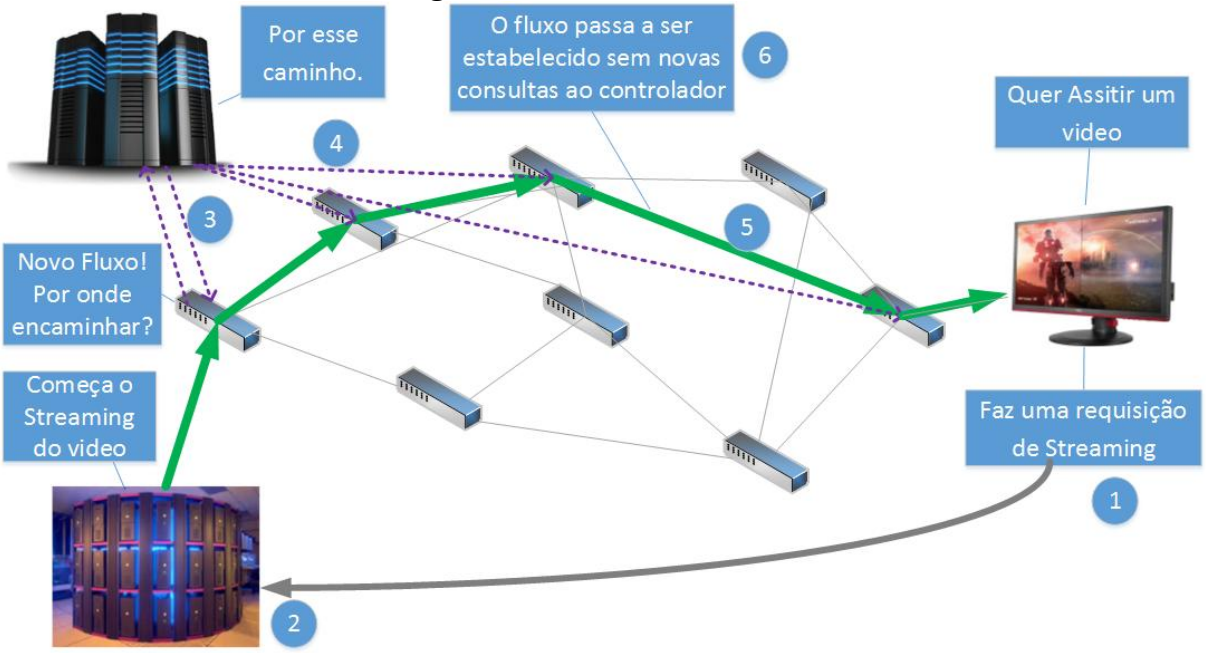
O plano de controle em SDN é a abstração que encapsula os conceitos relacionados ao controle de uma rede de computadores convencional. Nele encontram-se as tomadas de decisões, de tal modo que garanta o funcionamento da rede. Apesar de ser um elemento logicamente centralizado, não significa que seja apenas um equipamento. De maneira a garantir uma maior robustez na rede e garantia de um melhor desempenho é possível replicá-lo por diferentes dispositivos, ou seja, ser fisicamente distribuídos (Cardoso, 2015). Além disso, o *switch* pode estabelecer a comunicação com um único controlador ou pode fazer isso com vários controladores ao mesmo tempo. Isso aumenta a redundância e torna a rede mais tolerante a falhas.

Dentre as principais funções de um controlador se destaca a adição e remoção de fluxos nas tabelas dos *switches*. Na Figura 2 é possível acompanhar um exemplo do estabelecimento de um fluxo de acordo com os seguintes passos: 1º solicitação de algum conteúdo, como um vídeo,

por exemplo; 2º a solicitação é respondida e o fluxo é então encaminhado para o *switch*, que consulta as suas tabelas de fluxo e identifica um novo fluxo; 3º o *switch* envia uma solicitação para o controlador requisitando um caminho por onde ele deve encaminhar o novo fluxo; 4º o controlador, que possui uma visão geral da rede, responde ao *switch*, mostrando-lhe o caminho que o fluxo deve seguir; 5º o fluxo é encaminhado para os demais *switches* para estabelecer o fluxo; e 6º o fluxo é estabelecido sem novas consultas ao controlador.

Assim como as arquiteturas de computadores possuem diversas opções de sistemas operacionais como Windows, Linux, Mac, etc., em SDN também não é diferente. Existem inúmeros controladores SDN, alguns com semelhanças, mas outros com características peculiares, como tipo de protocolo API *southbound*, modularidade, etc. A maioria dos controladores suportam apenas o OpenFlow na API *southbound* e existem inúmeras APIs *northbound*, como APIs *ad-hoc*, APIs Restful, interfaces de programação multinível e sistemas de arquivos. Existem controladores com arquitetura distribuída ou centralizada, que são *multi-threaded*, mais consistentes e mais tolerantes a falhas, que suportem a versão do OpenFlow 1.0, 1.1, 1.2, 1.3, 1.4 ou 1.5, codificados em diversas linguagens de programação, tais como Java, C++, Python e C, e inúmeras outras características (Kreutz et al., 2015).

Figura 2 - Estabelecimento de fluxo.



Fonte: Adaptado de (Duque et al., 2012).

2.4.1 Nox

Nox (Gude et al., 2008) é um controlador OpenFlow desenvolvido inicialmente pela Nicira® e, a partir de 2008, disponibilizado para a comunidade. O Nox foi o primeiro controlador desenvolvido que deu base para o modelo SDN, juntamente com o protocolo OpenFlow. A interface de programação disponível para criar aplicações, assim como a linguagem utilizada no desenvolvimento do controlador em questão foi o C++. Uma das principais características deste controlador é o alto desempenho. A partir do controlador Nox surgiu um novo projeto, cujo objetivo era prover uma interface mais simples para controladores SDN. Desenvolvido em Python, o controlador Pox é uma versão do controlador Nox tradicional. Este controlador normalmente é utilizado como uma alternativa ao Nox em experimentos e prototipação, já que possui uma interface mais amigável.

2.4.2 OpenDayLight

O OpenDayLight (ODL) (Opendaylight, 2016) é um projeto *open source* cujo desenvolvimento tem participação de grandes empresas como Cisco®, Citrix®, Microsoft®, IBM®, dentre outras. O principal objetivo da comunidade é acelerar o processo de popularização do uso de SDN's e, como parte do projeto, disponibilizar a plataforma para o desenvolvimento de aplicações em SDN's. O OpenFlow é um dos padrões suportados pelo ODL, atualmente tem suporte às versões 1.0 e 1.3 do OpenFlow.

2.4.3 Ryu

Ryu (Ryu, 2014) é um controlador de código aberto desenvolvido por um grupo japonês da NTT Lab's. O projeto é totalmente implementado em Python, e possui boa integração com outras ferramentas de rede, por exemplo, o OpenStack (Corradi; Fanelli; Foschini, 2014). Um dos principais pontos do projeto é o suporte a vários protocolos de southbound, como OpenFlow, NetConf e OF-Config. O desenvolvimento constante por parte da comunidade permite que o controlador esteja atualizado com as versões mais recentes do OpenFlow. Atualmente, possui suporte total até a versão 1.4, e algumas implementações parciais da versão 1.5 do protocolo.

2.4.4 ONOS

ONOS (Open Network Operating System), (ONOS, 2014) é um controlador de código aberto desenvolvido por um grupo disponibilizado em dezembro de 2014 pelo ON.Lab. O projeto

é escrito em Java, usando OSGi para o gerenciamento de suas funcionalidades. Possui suporte a OpenFlow, NetConf, OpenConfig, etc. na *southbound*, porém sem estarem ligados diretamente a sua arquitetura. Como destaque, seu desenvolvimento foi norteado por requisitos de alto desempenho, disponibilidade, baixa latência e capacidade de manipulação de grandes redes, além de possuir a infraestrutura do núcleo modular e distribuída, sendo direcionado, principalmente, para prestadoras de serviços de missão crítica (Martiña, 2015).

2.4.5 Floodlight

O Floodlight é um controlador SDN de código aberto que opera na linguagem Java, e tem seu funcionamento baseado na operação de multithread. Composto por um conjunto de módulos, no qual cada um provê serviços para outros módulos e tem o controle através de uma API Java ou API REST. Foi desenvolvido para operar em alto desempenho, sendo o núcleo de uma grande empresa SDN, a *Big Switch Networks*. Além do mais, pode operar tanto com equipamentos que suportem OpenFlow quanto com os que não o suportem, além de gerenciar diversos conjuntos de equipamentos, tratando-os como várias "ilhas".

A versão mais recente do Floodlight é capaz de utilizar até a versão 1.4 do protocolo OpenFlow, porém já está em desenvolvimento a nova versão para o OpenFlow 1.5 (Floodlight, 2014). Originalmente, é um projeto derivado de outro controlador SDN chamado Beacon, baseado em Java, criado em 2010, muito utilizado para ensino e pesquisa (Erickson, 2013). Atualmente, é administrado pela ONF e pode ser executado nos diversos sistemas operacionais, tais como Linux, Mac OS e Windows.

O uso desse controlador é simples e possui uma vasta documentação, disponibilizando diversos exemplos úteis para o desenvolvimento de novas aplicações. Também possui um grupo de desenvolvedores ativo, facilitando no esclarecimento de dúvidas diretamente com a comunidade[2] envolvida no desenvolvimento do projeto Floodlight. Essas características motivaram a escolha desse controlador para o desenvolvimento do presente trabalho.

[2] https://groups.google.com/a/openflowhub.org/forum/#!forum/floodlight-dev

2.5 Plano de Dados

O plano de dados em uma rede definida por *software* tem como função central o encaminhamento de pacotes. Logo, o plano de dados passa a ser representado pelos equipamentos de rede, sejam físicos ou virtuais, que possuam a capacidade de encaminhar pacotes.

Dentre os equipamentos virtuais está o OVS[3] (Open Virtual Switch), que é um switch virtual multicamada licenciado sob a licença Open Source Apache 2.0. Ele é projetado para permitir a automatização maciça da rede por meio da extensão programática, enquanto que ainda suporta interfaces e protocolos de gerenciamento padrão (por exemplo, NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag). Além disso, ele é projetado para oferecer suporte à distribuição em vários servidores físicos semelhantes, como o vNetwork distribuído pela VMware® e o vSwitch ou o Nexus 1000V da Cisco®.

O OpenvSwitch atua como uma ponte que balanceia o tráfego de uma aplicação através de várias interfaces. Em sua abordagem, um aplicativo envia seus dados para uma interface ethernet virtual, a qual está liga a várias interfaces físicas. A pilha de rede se encarrega de balancear o tráfego por meio múltiplas interfaces (Navarro et al., 2015).

2.5.1 *Switch* OpenFlow

Esta seção descreve os requisitos de um *Switch Logical* OpenFlow, abrangendo os componentes e as funções básicas do *switch* OpenFlow.

Um destaque para a arquitetura do *switch* OpenFlow 1.5 é a existência de várias tabelas de fluxo. Assim como os processadores de computador podem fazer uso de *pipelining*, os *switches* podem executar pesquisas e processamento de cabeçalho. Um documento muito legível que justifica a necessidade e exemplos de utilização de múltiplas tabelas é o ONF TR-510 sobre *The Benefits of Multiple Flow Tables and TTPs*[4], destacando-se a comunicação com vários controladores que possibilitam maior resiliência. Além desses componentes, o *switch* tem um conjunto de portas, a tabela de grupos e a tabela de métricas.

---

[3] http://openvswitch.org/

[4]https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_Multiple_Flow_Tables_and_TTPs.pdf

2.5.1.1 Portas

As portas OpenFlow são as interfaces de rede para a passagem de pacotes entre o processamento OpenFlow e o restante da rede. Os *switches* OpenFlow se conectam logicamente através de suas portas OpenFlow; um pacote pode ser encaminhado de um *switch* OpenFlow para outro, somente pela porta OpenFlow de saída no primeiro *switch* e uma entrada OpenFlow, no segundo *switch*.

2.5.1.2 Tabelas de fluxo

Uma tabela de fluxo *flow-table* consiste em um banco de dados de entradas de fluxos *flow-entries* com alguns componentes principais, nos quais algumas ações podem ser destacadas: encaminhar o pacote para uma porta específica do *switch*, descartá-lo, encapsulá-lo e criptografa-lo, limitar a banda, etc.

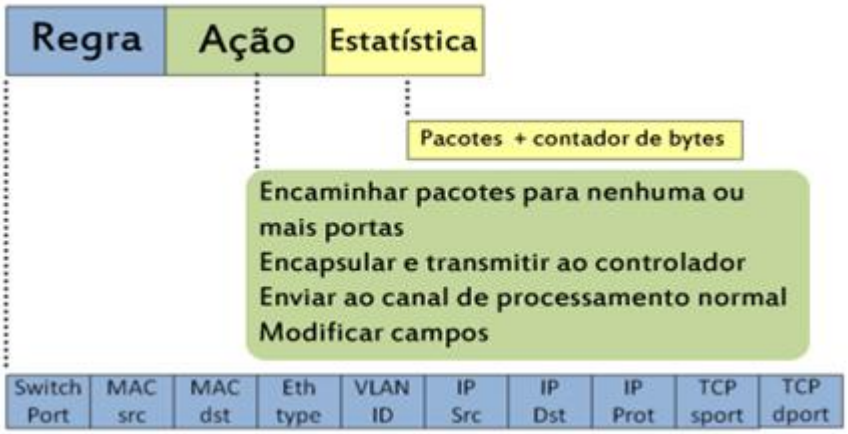
Cada entrada de fluxo possui três campos (Figura 3) que mostram ao *switch* como proceder a cada fluxo:

Regra - Consiste em fazer uma correspondência, um *match*, entre os campos nos cabeçalhos dos pacotes. Esta regra faz com que os fluxos sejam definidos, ou seja, os pacotes serão classificados como parte de um fluxo.

Ação - Define o processamento dos pacotes.

Estatísticas - Faz o registro dos números de pacotes e bytes que serão dedicados a cada fluxo e o tempo transcorrido desde que o último pacote fez o *match*, auxiliando na remoção de fluxos inativos.

Figura 3 - Tabela de fluxo.



Fonte: Extraído de (Duque et al., 2012).

Como citado anteriormente, cada entrada de fluxo da tabela de fluxo apresenta um conjunto de ações correspondentes que permitem encaminhar os pacotes, modificar os campos, ou descartar o pacote. Assim, quando um *switch* recebe um pacote que não possui uma entrada em sua tabela de fluxos, ele envia esse pacote para o controlador. Este, então, toma as decisões de como lidar com o pacote, podendo descartá-lo ou adicionar uma entrada de fluxo para encaminhar pacotes semelhantes no futuro.

2.5.1.3 Canal de processamento - *Pipeline Processing*

Cada *switch* possui múltiplas tabelas de fluxo e cada tabela de fluxo possui múltiplos fluxos de entrada. O canal de processamento define como os pacotes interagem com essas tabelas.

2.5.1.4 Tabela de grupos - *Group table*

O item tabela de grupos pertencente ao *switch* OpenFlow foi adicionado na versão 1.1 do protocolo, para proporcionar novas formas de encaminhamento. Cada grupo possui um conjunto de ações chamado *Actions Buckets*. Os pacotes chegam à tabela de grupos depois de um pacote ser processado por uma ação de grupo (Fernandes; Rothenberg, 2014).

Actions Buckets são listas ordenadas de *buckets* de ação, nas quais cada *buck* de ação contém um conjunto de ações para executar. As ações em um *buck* são sempre aplicadas como um conjunto de ação.

2.5.1.5 Tabela de métricas - *Meter Table*

Uma tabela de métricas consiste em entradas de métricas definidas por fluxo, permitindo que o OpenFlow implemente várias operações de QoS simples, como limitação de taxa. Pode ser combinado com filas através de porta para implementar estruturas complexas de QoS, como DiffServ.

Uma métrica avalia a taxa de pacotes atribuídos a ela e permite controlar a taxa desses pacotes. As métricas são conectadas diretamente às entradas de fluxo (ao contrário das filas que são anexadas às portas). Qualquer entrada de fluxo pode especificar uma medida em uma lista de ações.

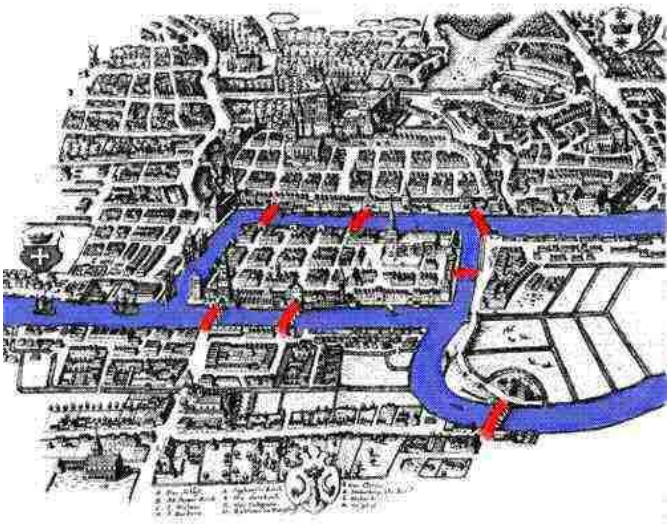
2.6 Algoritmos de Busca de Caminho

A teoria de grafos é uma das ferramentas matemáticas mais simples e poderosas usadas para construção de modelos e resolução de problemas, por exemplo: fluxos em redes, escolha de uma rota ótima, tática e logística.

A mais antiga citação sobre a teoria ocorreu no ano de 1735, pelo matemático suíço Leonhard Euler, em seu artigo “The Seven Bridges of Königsberg” (Melo, 2016).

No rio Pregel, que corta a cidade de Königsberg (região da Prússia), apresentado na Figura 4, havia duas ilhas que, na época, eram ligadas entre si por uma ponte. As duas ilhas se ligavam ainda às margens por mais seis pontes. Dizia-se que os habitantes da cidade, nos dias soalheiros de descanso, tentavam efetuar um percurso que os obrigasse a passar por todas as pontes, mas apenas uma vez em cada uma. Como as suas tentativas foram sempre falhas, a pergunta era: é possível passar pelas sete pontes numa caminhada contínua sem passar duas vezes por qualquer uma das pontes?

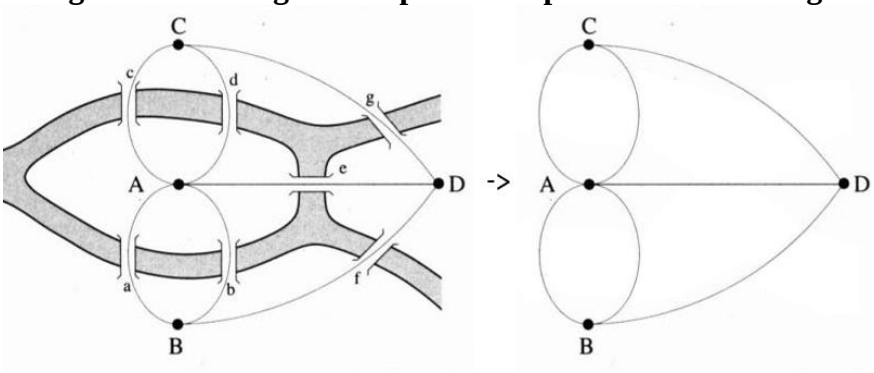
Figura 4 - Pontes de Könisberg.



Fonte: Extraído de (Araújo, 2016).

Levando em consideração que as porções de terra seriam consideradas como vértices e as pontes como arestas, teríamos o grafo descrito na Figura 5.

Figura 5 - Grafo gerado a partir das pontes de Könisberg.



Fonte: Adaptado de (Carroll, 2009).

Euler analisou o problema trocando as áreas de terra por pontos (nós) e as pontes por arcos (ramos), modelando. Desta forma, provou que o problema não tinha solução - uma análise detalhada de como Euler chegou a esta conclusão pode ser encontrado em Rabuske (1992). Teve início, assim, o que hoje conhecemos como Teoria de Grafos.

Com a teoria dos grafos foi possível estudar e desenvolver soluções para melhoramento de percursos, desenvolvendo algoritmos que pudessem mostrar os melhores caminhos para se chegar a um ponto, tendo como referências as distâncias ou números de saltos. Assim vários algoritmos de busca foram desenvolvidos para descobrir percursos entre dois pontos, onde cada um possui o seu referencial ou especialidade.

2.6.1 Descrição de algoritmos de busca

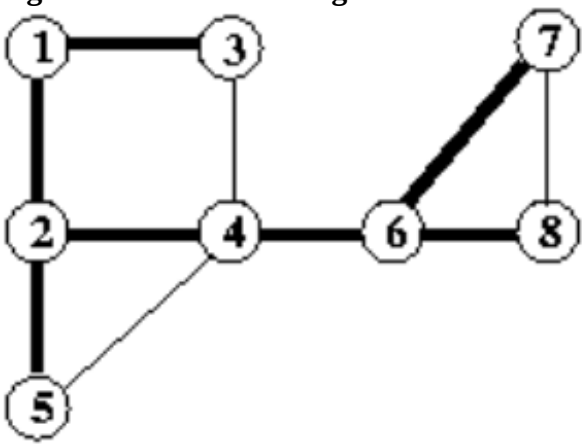
Como visto, a teoria dos grafos veio com a finalidade de explicar e solucionar problemas nos quais se tem intercomunicação entre objetos, lugares e etc., ou seja, algo que venha a formar um grafo. Além disso, há vários algoritmos que visam encontrar caminhos em grafos; nesta seção serão descritos dois destes algoritmos.

2.6.2 Algoritmo de busca em largura

O algoritmo de busca em largura, também conhecido como *breadth-first search* (*BFS),* é um método de busca que expande e examina sistematicamente todos os nós de um grafo em busca de uma solução. Pode-se dizer, pois, que este algoritmo realiza uma busca exaustiva em um grafo inteiro, sem considerar o seu alvo de busca até que ele o encontre. A busca em largura começa por um vértice, que pode ser especificado pelo usuário. Depois o algoritmo visita todos os vértices que estão à distância 1 de s;em seguida, todos os vértices que estão à distância 2 de s, e assim

sucessivamente. Para implementar tal ideia, o algoritmo usa uma fila de vértices, a qual contém todos os vértices já visitados, cujos vizinhos ainda não foram visitados.

Figura 6 - Busca em Largura em um Grafo.



Fonte: Extraído de (Gagnon, 2001)

Considerando que todos os enlaces possuem o mesmo peso, o algoritmo possui eficiência ótima, pois o mesmo irá encontrar a melhor solução com o menor número de saltos (passagem de um vértice para outro).

Utilizando o algoritmo BFS no gráfico representado na Figura 6, é possível encontrar o menor caminho entre todos os vértices em relação ao vértice inicial. No grafo estão representados todos os caminhos mais próximos, em negrito.

O algoritmo funciona da seguinte maneira: inicia com o vértice de número 1, a partir dele verifica-se quais os demais nós são alcançados, neste caso o número 2 e 3; ele insere esses vértices em uma fila, finalizando a verificação no vértice 1. Segue para o próximo elemento da fila, neste caso o vértice de número 2, e, novamente, verifica quais os vértices são alcançados por ele, inserindo-os na fila; caso o vértice de número 2 esteja ligado a algum vértice já existente na fila, ele o ignora, pois como já se encontra na fila, significa que o menor caminho até ele já foi encontrado. O algoritmo se repete até que encontre todos os vértices com menor caminho ou encontre seu objetivo. No exemplo ilustrado na Figura 6, o objetivo foi encontrar o menor caminho entre o vértice inicial e os demais.

2.6.3 Algoritmo de Dijkstra

O algoritmo de Dijkstra foi desenvolvido por Edsger Wybe Dijkstra, em 1959, tendo por função encontrar o caminho mais curto entre duas arestas dentro de um grafo. Para definição deste caminho mínimo é necessário que cada aresta possua um peso positivo, ou seja, cada aresta deve

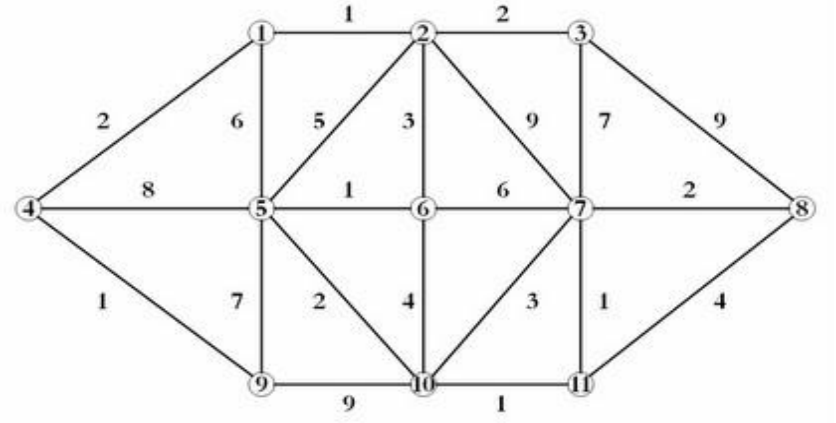
possuir uma métrica para que o algoritmo possa encontrar um caminho, cuja soma dos valores (pesos) das arestas seja o menor, comparado com as outras possibilidades de caminhos (Barros; Pamboukian; Zamboni, 2007).

Dentro de uma rede, para um melhor desempenho do serviço de roteamento, é extremamente necessária a definição de uma métrica que gerará um valor para cada nó pertencente à rede. Esta métrica será o valor utilizado pelo qual o algoritmo de Dijsktra define o caminho de menor custo entre o ponto inicial e o ponto final. Poderá ser qualquer valor, como congestionamento, taxa de perda, atraso do enlace ou velocidade.

Após a definição do valor que será utilizado para a escolha do caminho de menor custo, barros; Pamboukian; Zamboni (2007) explicam que o algoritmo de Dijkstra inicia-se verificando todos os custos dos vértices disponíveis em um grafo, conforme visualizado na Figura 7.

Após a definição dos custos, são levantados os valores dos vizinhos do nó inicial. Se o ponto final for um vizinho, o custo do enlace até o destino é o valor entre os nós, caso contrário o custo do enlace é considerado como infinito.

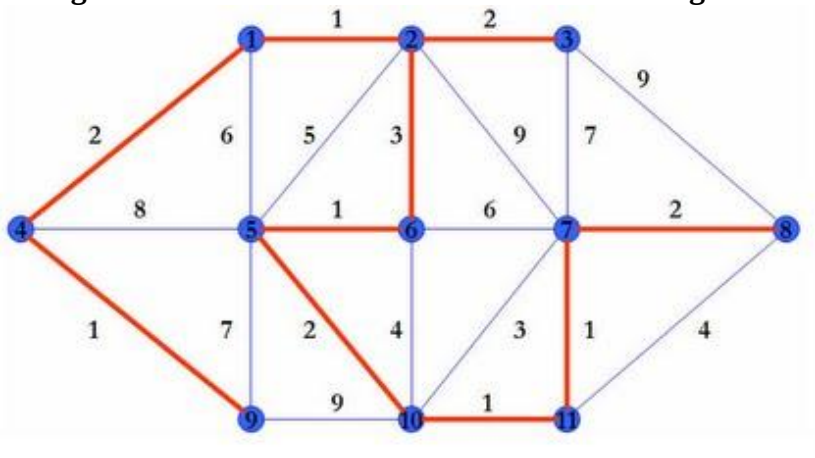
Figura 7 - Algoritmo de Dijkstra: definição dos custos dos vértices.



Fonte: Extraído de (Barros; Pamboukian; Zamboni, 2007)

Com o valor de custo definido como infinito, o vizinho de menor valor passa a ser considerado como ponto inicial e os seus vizinhos são analisados iniciando-se um *loop*, que verificará todos os vértices até que sejam levantados os caminhos de custos mais baixos. A Figura 8 representa os caminhos mais curtos entre todos os vértices existentes no grafo.

Figura 8 - Caminhos mais curtos dentro de um grafo.



Fonte: Extraído de (Barros; Pamboukian; Zamboni, 2007).

O algoritmo de Dijkstra finaliza o seu processo quando o número total dos caminhos curtos pesquisados for igual ao número de vértices disponibilizados no grafo.

2.6.4 Comparação entre os Algoritmos

Os algoritmos apresentados são exemplos de soluções de busca de caminhos em grafos, porém, o algoritmo de Dijkstra e o algoritmo de busca em largura, apesar de encontrarem caminhos entre dois pontos de um grafo, possuem condições diferentes para encontrarem um caminho. Como se percebe, a busca em largura considera todos os valores de arestas como se fossem iguais, enquanto que o algoritmo de Dijkstra considera valores para as arestas, efetuando a busca sempre em busca do menor valor. Assim, este último enquadra-se melhor para buscas em redes de computadores.

2.7 Conclusões do Capítulo

Este capítulo apresentou conceitos básicos sobre a arquitetura TCP/IP, bem como sobre o paradigma de redes SDN, apresentando suas características e uma descrição mais detalhada sobre SDN. Além disso, foram citados os principais controladores, bem como o funcionamento do protocolo a OpenFlow com os principais elementos pertencentes a um *switch* OpenFlow. O capítulo também apresenta um breve histórico sobre buscas de caminhos, destacando dois algoritmos muito consagrados na literatura, sendo um deles a base para o desenvolvimento da proposta deste trabalho.

3 TRABALHOS RELACIONADOS

O objetivo deste capítulo é apresentar um conjunto de trabalhos e pesquisas associados ao tema desta dissertação, bem como relacioná-los ou compará-los ao trabalho proposto. Durante a etapa de levantamento bibliográfico, muitos outros trabalhos, não descritos neste capítulo, foram investigados. Portanto, este capítulo visa fornecer um estudo detalhado daqueles trabalhos que serviram para fundamentar conceitos, alicerçar tomadas de decisões ou ainda delimitar o escopo de ações acerca da proposta.

Brito et al., (2014) apresenta as vantagens do uso de SDN em uma rede *Mesh* sem fio. Dentre elas, destaca-se a possibilidade de buscas de caminhos baseados em parâmetros de rede, tais como lista de associações, largura de banda projetada, relação sinal/ruído e banda residual. O autor sugere a possibilidade de combinação dessas métricas para gerar um peso e usar no algoritmo de Dijkstra tradicional, porém não apresenta como essa combinação de métricas pode ser feita.

Jiang et al. (2014) desenvolveu uma proposta semelhante à descrita no presente estudo, na qual o autor sugere uma modificação no algoritmo de Dijkstra, objetivando considerar mais de uma variável na busca. Além do valor considerado como peso para aresta, a proposta também utiliza um valor para cada vértice, desta forma, ao invés de comparar apenas o valor da aresta na condição, ele efetua uma soma dos valores de peso da aresta mais o peso do vértice. O que difere desta proposta são as variáveis utilizadas na busca, bem como a forma de combinação entre elas. A proposta apenas inclui mais um variável na condição do algoritmo de Dijkstra para considerar um valor para o vértice, ou seja, a condição depende do peso da aresta, dada pela largura de banda, mais o peso do vértice, dado pela capacidade do *switch*.

Tomovic; Prasad; Radusinovic (2014) apresenta um framework para aprovisionamento de QoS em SDN. Dentre os módulos propostos, destaca-se o de cálculo de rota, que efetua a busca utilizando o algoritmo de Dijkstra. Diferentemente dos trabalhos anteriores, apesar de efetuar a busca utilizando a largura de banda como peso para os enlaces, a proposta considera apenas o subgrafo que possui enlaces com capacidade suficiente para a banda solicitada. Neste caso, a proposta faz a busca sempre verificando a banda disponível, sendo essa o resultado de um cálculo envolvendo a banda utilizada e a reservada. A proposta reduz o grafo nas decisões do algoritmo, porém o único fator considerado foi a largura de banda, pois o objetivo é encontrar um caminho

com banda que atenda a solicitação de QoS, porém não utiliza as demais possibilidades de reduzir o grafo por meio de outros parâmetros de QoS como latência, perda de pacotes, etc.

Além dos trabalhos anteriormente citados, inúmeros outros utilizam o algoritmo de Dijkstra para efetuarem suas buscas, como Tairaku et al. (2016), Raghavendra; Lobro; Lee (2012), Cohen et al. (2014) que, em geral, mudam o parâmetro de busca, utilizando, na maioria dos casos, atraso do enlace ou largura de banda, porém sempre baseados em uma única métrica.

Diferentemente dos trabalhos anteriores, Rocha; Verdi (2015), não utilizam o algoritmo de Dijkstra. Em seus trabalhos apresentam um conjunto de ferramentas chamadas MILPFlow (*Mixed Integer Linear Programming with OpenFlow*), para gerar modelos computacionais de *datacenters* para resolver problemas de roteamento. Nesse caso, o autor apresenta uma solução para busca de caminhos utilizando o algoritmo de busca em profundidade (DFS - *depth-first search*). O uso desse algoritmo se diferencia do algoritmo de Dijkstra, porque não utiliza os parâmetros de QoS da rede, pois a proposta leva em consideração apenas o número de saltos para encontrar o destino.

Muitos trabalhos utilizam o algoritmo de Dijkstra sem modificações, apenas inserindo parâmetros diferentes nas condições de busca. Dentre os trabalhos encontrados, apenas um sugere o uso de dois parâmetros, porém a proposta encontrada apenas inclui mais um parâmetro na mesma condição de busca do algoritmo. Neste trabalho sugerimos uma modificação no algoritmo de Dijkstra, incluindo o uso de duas condições que servem como barreiras para direcionar o algoritmo, sendo elas um limite máximo para latência e perda de pacotes. Além desses, utilizamos, como condição de busca o atraso de cada enlace. O diferencial no trabalho proposto encontra-se na forma como a busca é efetuada, feita apenas no subgrafo que atenda as condições limítrofes, sendo que este subgrafo é gerado em tempo de execução, assim é possível oferecer mais performance ao algoritmo.

Em Rezende (2016), apresenta um mecanismo para prover roteamento multicaminhos em Redes OpenFlow. Tratou-se de uma proposta para ramificar fluxos na rede, objetivando atender demandas de largura de banda. Utilizando módulos de monitoramento apresentados em trabalhos anteriores do próprio autor é possível coletar informações da rede e efetuar buscas de caminhos até que se encontre um conjunto de caminhos capazes de compor a um SLA solicitado. A proposta faz uso do algoritmo de Dijkstra, porém sugerindo o uso de inúmeras buscas, usando esse algoritmo sem modificações, apenas utilizando como peso para as arestas a largura de banda da rede. Considerando a possibilidade da coleta de vários outros dados, tal como é citado pelo autor, seria possível aproveitar esses dados para efetuar a busca dos múltiplos caminhos.

Em Sandri et al. (2015) propõe o uso do TCP de caminho múltiplo (MPTCP - *MultiPath* TCP) em conjunto com a arquitetura SDN. Na abordagem proposta pelos autores, um controlador SDN é utilizado para garantir que os subfluxos gerados pelo MPTCP sempre passem por caminhos disjuntos. O *host* define primeiramente quantos subfluxos serão usados pelo MPTCP e com o algoritmo proposto é possível encontrar os caminhos que atendam ao SLA solicitado. Pode-se destacar, nesse trabalho que, apesar de ser proposto um algoritmo que encontre vários caminhos disjuntos, o autor inclui, no interior de seu algoritmo, o uso de Dijkstra para encontrar cada caminho; a cada caminho encontrado o autor passa novamente ao algoritmo de origem, destino e a topologia a qual está sendo feita a busca, porém, esta, com as modificações efetuadas na busca anterior. O que se pode destacar, em relação a proposta desta dissertação, é que mais uma vez o algoritmo de Dijkstra foi utilizado sem nenhuma modificação, recaindo a modelos de busca antigos sem aproveitar o conhecimento adquirido pelo controlador para acelerar a busca.

Além desses, podemos citar Yan et al. (2015), Sandri; Silva; Verdi (2015), dentro outros, que também trabalham com a ideia de múltiplas rotas, porém mais uma vez utilizando o algoritmo de Dijkstra nas buscas de caminhos, utilizando um dos parâmetros de QoS, na maioria dos casos a largura de banda.

Como se pode notar pela quantidade de trabalhos citados, o algoritmo de Dijkstra é consagrado na literatura e utilizado pela grande maioria dos autores ao efetuarem buscas de caminhos. Uma vez que em SDN o controlador possui um conhecimento dos parâmetros de QoS da rede e os autores da literatura utilizam alguns desses parâmetros para efetuarem suas buscas utilizando o algoritmo de Dijkstra. A Tabela 1 apresenta uma comparação entre os trabalhos apresentados com os respectivos parâmetros utilizados por cada algoritmo.

Tabela 1 - Comparativo dos parâmetros utilizados pelos trabalhos relacionados.

	Largura de banda	Atraso	Perda de pacotes	Relação sinal/ruído	Latência	Dijkstra
Brito et al., (2014)	X			X		X
Tomovic; Prasad; Radusinovic (2014)	X					X
Yan et al. (2015)		X				X
Sandri; Silva; Verdi (2015),	X					X
Tairaku et al. (2016),	X					X
Q-Path		X	X		X	X

Fonte: Elaborada pelo autor.

Assim considerado, o trabalho apresentado nesta dissertação propõe um aproveitamento do conhecimento existente no controlador para combinar métricas de QoS no algoritmo de Dijkstra, uma vez que se apresenta como mais utilizado e já consagrado, melhor algoritmo de busca de menor caminho. Desta forma, avaliamos a estrutura de funcionamento do algoritmo de Dijkstra e o modificamos para criar o algoritmo proposto neste trabalho, o Q-Path.

4 PROPOSTA

Este capítulo apresenta o Q-Path, uma proposta de algoritmo de busca de caminhos utilizando o conhecimento dos recursos de uma rede SDN. Além deste algoritmo a proposta apresenta um módulo no controlador ao qual foi implementado. Para alcançar o objetivo, há a realização do monitoramento da rede, via navegador; e a interação a interação com o usuário, possibilitando a inserção da configuração de caminho almejada. Além disso, apresenta uma seção de trabalhos relacionados ao Q-Path.

4.1 Visão Geral e Escopo da Proposta

Q-Path é uma proposta de algoritmo de busca de caminho desenvolvido de forma direta para o paradigma SDN - devido este possuir um conhecimento global da infraestrutura - em busca de oferecer QoS. Dessa forma, é possível aproveitar esse conhecimento dos recursos da rede para encontrar o caminho entre dois nós da rede de forma mais ágil. Mediante esta diretriz, o nome da proposta apresenta uma abreviação a sigla “QoS” combinada com a palavra “Path” do inglês “caminho”, “curso” ou “trajetória”.

O algoritmo tem como premissa o conhecimento da origem e destino ao qual se deseja a interconexão, bem como responsável pelo atraso de cada enlace. Além disso, é possível a inclusão de barreiras ou condições que podem influenciar o desempenho. Tais barreiras podem ser: valor máximo de perda de pacotes nas portas, latência, capacidade do enlace, custo do enlace, ou outra métrica que influencia na escolha de um caminho com QoS. No escopo deste trabalho são utilizadas apenas duas dessas barreiras, sendo elas: latência em milissegundos e valor máximo de perda de pacotes em uma porta. O parâmetro latência é um requisito importante para aplicações de tempo real ou sensíveis ao atraso. Enquanto que a perda de pacotes é um parâmetro importante para aplicações não elásticas. Em outras palavras, são dois parâmetros de QoS exigidos pelas aplicações que trafegam atualmente na Internet. Portanto, Q-Path considera esses dois parâmetros para tornar o algoritmo mais flexível a inúmeros padrões de tráfego.

Ademais, ressalta-se o desenvolvimento de um módulo no controlador para enquadrar o algoritmo proposto. Além desse, outros módulos são incrementados a arquitetura do controlador Floodligth para melhor comunicação com o núcleo da rede, bem como para coleta de informações. Algumas dessas comunicações ocorrem por meio de API disponibilizadas pelo controlador Floodligth.

Para possibilitar que um administrador de redes tenha a capacidade de gerenciar uma infraestrutura de redes utilizando o Q-Path, foi desenvolvida uma *interface* de interação com o usuário que pode ser acessada via navegador. Assim é possível solicitar buscas específicas de caminhos baseando-se em requisitos de QoS. Desta maneira, além de solicitações de caminho efetuadas diretamente pelo administrador da rede, a proposta permite a solicitação de caminhos por meio de aplicações, caso o administrador configure previamente qual tratamento deve ser feito com cada tipo de fluxo específico, desta forma o algoritmo pode encontrar caminhos baseados nessas configurações prévias.

Além disso, a proposta apresenta um passo a passo de uma execução do algoritmo visando facilitar o entendimento do leitor. Além disso, apresenta um pseudocódigo do algoritmo com um esclarecimento detalhado.

4.2 Algoritmo Q-Path

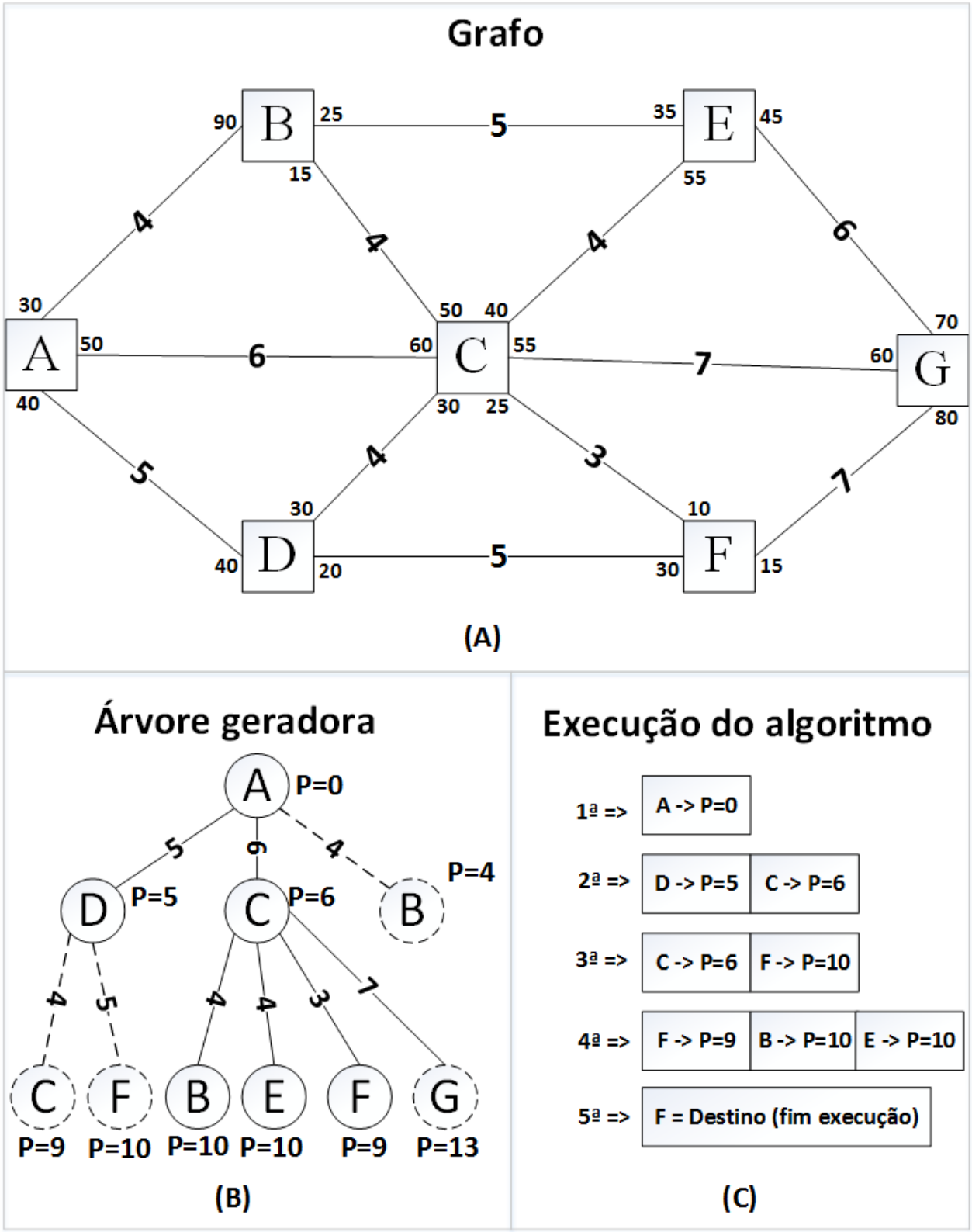
O algoritmo proposto é baseado no algoritmo de Dijkstra, tendo sido escolhido por ser um algoritmo consagrado na literatura como o melhor, quando o objetivo é encontrar o menor caminho partindo de uma origem para todos os demais vértices em um grafo ponderado.

Diante do exposto e considerando que as redes SDN são facilmente representadas por meio de grafos segundo Pantuza (PANTUZA et al., 2014), as características dos enlaces como atraso, custo, etc, podem ser usadas como pesos para as arestas do grafo, e os *switchs* ou roteadores tornam-se os vértices.

A Figura 9. a apresenta uma topologia de rede em forma de grafo. O exemplo ilustrado possui sete vértices que estão identificados por meio de letras de A à G. A topologia possui doze enlaces e cada um possui um tempo médio de atraso em milissegundos. Além desses, nas portas de cada enlace há um valor de perdas de pacotes correspondente.

Ao simular a execução da busca, a Figura 9.b apresenta uma possível árvore geradora mínima, porém nem todos os vértices do grafo encontram-se na árvore; isso ocorre porque uma das condições da busca é a existência de uma latência máxima a ser percorrida, ou seja, apenas os vértices pertencentes ao raio de distância delimitado pela latência pertencerão à busca e, consequentemente, à arvore geradora mínima.

Figura 9 – Algoritmo Q-Path com árvore geradora e execução do algoritmo.



Fonte: **Elaborada pelo autor.**

Considerando a perda de pacotes com valor máximo permitido de 70%, nota-se que o vértice B não é incluído inicialmente na árvore por possuir um valor superior ao delimitado. Paralelamente à Figura 9.c, nota-se que ao inserir o vértice raiz na árvore, o mesmo vértice também é inserido na lista de execução do algoritmo; isso ocorre porque a arvore é criada com base na execução do algoritmo.

Na Figura 9.c, a execução do algoritmo é apresentada em etapas objetivando facilitar o entendimento. Na 1ª etapa, é inserido na lista o vértice que iniciará o caminho; esse também será o nó raiz da árvore (Figura 9.b), como dito anteriormente. Na etapa 2, o vértice de menor valor é retirado da lista, ou seja, vértice A. Em seguida, todos os vértices alcançáveis, a partir de A, são inseridos na lista, desde que estejam enquadrados nas condições. Essas são: taxa de perda de pacotes máximo (definido anteriormente) e latência máxima a qual o algoritmo pode percorrer, que será definida no valor de 11 milissegundos, ou seja, esta busca considerará apenas os vértices alcançáveis com distância, a partir da origem, inferior ao 11 milissegundos. Nessas condições, nota-se que o vértice B não é incluído na lista de execução por possuir perda de pacotes superior ao aceitável.

Na etapa 3, nota-se que o vértice D já não é encontrado na lista devido estar sendo pesquisado no momento; ele foi escolhido por ser o de menor valor. Por se tratar de um algoritmo guloso (sempre escolhe vértices de menor distância), sempre efetuará a busca pelo vértice de menor valor. Os próximos nós alcançáveis, a partir de D, são C e F. Observa-se que C já está na lista com valor de distância da origem inferior a atual, ou seja, C está com valor 6; o novo caminho, passando por D até chegar a C, seria igual a 9, então C não sofre alteração na árvore geradora. Em contrapartida, F, que ainda não está na lista, passa a ser incluído assim como na árvore.

Na 4ª etapa o vértice corrente passa a ser o vértice C, por ser o de menor valor na lista. Com esse vértice é possível alcançar B que não foi incluído anteriormente na lista por possuir perda de pacotes superior ao limitado; neste momento, B passa a ser incluído por estar dentro das condições, porém com uma rota passando por C. O vértice E também passa a ser incluído na lista, bem como na árvore, por se enquadrar nas condições limítrofes.

Constata-se que o vértice F já está na lista, contudo neste momento passa a ser alcançado por uma nova rota com valor de distância da origem inferior ao atual, ou seja, está na lista com valor de distância 10, passando a ser alcançado (a partir de C) com distância 9; logo, o valor de F, na lista, passa a ser alterado e seu antecessor passa a ser C, ocorrendo uma migração de F na árvore, saindo de D e passando a pertencer a C. Além desses vértices, C também alcança G, mas

este não atende ao requisito de latência inferior a 11 milissegundos, logo não é incluído na lista, consequentemente, não será adicionado na árvore.

Finalmente, a etapa 5 encerra a execução do algoritmo. O vértice F, que é o destino, passa a ser alcançado. Por se tratar de um algoritmo guloso, conclui-se que esse é o menor caminho. A partir desse ponto, o restante da lista não é necessário, pois o destino já foi encontrado.

Diante do observado durante a aplicação do algoritmo, observamos que nem todos os vértices serão incluídos na lista, além do que, a lista é preenchida de forma gradual, desta forma reduz-se o tempo de ordenação ou busca pelo menor valor na lista. Ademais, em casos hipotéticos, alguns vértices do grafo podem não estar dentro da área limítrofe (como ocorreu com o Vértice G), levando a uma redução da topologia em tempo de execução, consequentemente tornando a busca mais ágil.

Apesar das melhorias em tempo de execução, a complexidade do algoritmo no pior caso (quando nenhuma das condições sejam atingidas), iguala-se ao algoritmo de Dijkstra. Considerando Dijkstra com uma complexidade como $O((|V| + |A|) \log 2 |V|)$, explanado em GONÇALVES (2015), pode-se assumir tal complexidade para Q-Path. Porém, apesar de tal semelhança, podemos afirmar que, considerando $f(n)$, para Q-Path, e $g(n)$, para Dijkstra, $f(n) \in O(g(n))$, pois Q-Path possui uma ordem de crescimento igual ou inferior à ordem de crescimento de Dijkstra.

4.3 Descrição do Código do Algoritmo

Na seção anterior foi apresentada a descrição do funcionamento do Q-Path. Nesta seção, será exposta uma descrição do pseudocódigo do algoritmo de forma que se possa implementá-lo em qualquer linguagem de programação.

O algoritmo de busca é apresentado na Tabela 2, utilizando o formalismo matemático.

Tabela 2. Q-Path.

ALGORITMO Q-Path

```
1 - para todo v ∈ V[G]
2 -   d[v]←∞
3 - π [s] ←-1
4 - d[s] ←0
5 - Q ←s
6 - enquanto Q ≠ ø
7 -   u ←extrair-mín(Q)                    //Q ←Q - {u}
```

```
8 -   se u = r
9 -         então FIM
10 -  para cada v adjacente a u
11 -              se perdaPacotes(v) < P e v ≠ visitado
12 -              se d[u] + w(u, v) < L
13 -                    então Q ←v
14 -                    se d[v] > d[u] + w(u, v)          //relaxe (u, v)
15 -                          então d[v] ←d[u] + w(u, v)
16 -                                      π[v] ←u
```

Fonte: Elaborada pelo autor.

No qual:

G: grafo;

v: vértice qualquer do grafo;

s: vértice origem;

r: vértice destino;

u: pivô, vértice escolhido com peso mínimo na fila Q;

Q: lista de vértices a serem pesquisados;

w(u, v): custo da aresta, partido de u ao vértice vizinho v;

V[G]: conjunto de vértices (v) que formam o grafo G;

d[v]: custo caminho entre a origem e o vértice v;

π[v]: caminho entre a origem e o vértice v;

L: latência máxima determinada pelo administrador.

P: Taxa máximo de perda de pacotes aceitável pelo administrador.

Objetivando melhor entendimento, o algoritmo será apresentado em três partes listadas a seguir:

1º Parte: Iniciam-se os vetores:

```
1 - para todo v ∈ V[G]
2 -   d[v]←∞
3 - π [s] ←-1
4 - d[s] ←0
```

A primeira parte do algoritmo é importante para a inicialização das variáveis. Nela todos os vértices do grafo são identificados no conjunto de vértices do grafo com valor de distância da origem como infinito, exceto o vértice de origem que é inicializado com valor zero, devido não haver distância definida para ele. Como o vértice s simboliza a origem do caminho e não há

nenhum vértice antes dele, sendo que o vetor é identificado como -1, utilizado como condição de parada na geração do caminho.

2º Parte: conjunto de vértices a serem pesquisados.

```
5 - Q ← s
```

O grande diferencial deste algoritmo encontra-se no tamanho de *Q*. No algoritmo de Dijkstra, por exemplo, todos os vértices do grafo são inseridos inicialmente em *Q* e a cada interação ocorre uma busca pelo vértice de menor peso dentre todos os pertencentes à lista. Quanto maior o tamanho da lista, mais tempo é necessário para se encontrar o vértice desejado. Neste trabalho, objetiva-se manter *Q* da forma mais enxuta possível, pois apenas os vértices que realmente serão pesquisados irão ser inseridos em *Q*. Ao invés de iniciar com todos os vértices do grafo, tal como Dijkstra, em Q-Path, *Q* inicializa apenas com a origem e somente os vértices alcançáveis (dentro das condições impostas pelo administrador), a partir de *u* são inseridos em *Q*.

3º Parte: relaxamento das arestas

```
6 - enquanto Q ≠ ø
7 -   u ←extrair-mín(Q)                        //Q ←Q - {u}
8 -   se u = r
9 -         então FIM
10 -  para cada v adjacente a u
11 -              se perdaPacotes(v) < P e v ≠ visitado
12 -              se d[u] + w(u, v) < L
13 -                    então Q ←v
14 -                    se d[v] > d[u] + w(u, v)          //relaxe (u, v)
15 -                          então d[v] ←d[u] + w(u, v)
16 -                                      π[v] ←u
```

A terceira parte apresenta o laço que levará às interações do algoritmo, sendo que este possui duas condições de parada. A primeira ocorre quando o conjunto de vértices *Q* está vazio (linha 6), ou seja, não há mais vértices a serem pesquisados, concluindo que não há caminho entre a origem e o destino para as condições passadas. A segunda condição ocorre no momento em que o vértice destino foi encontrado (linha 8), por se tratar de um algoritmo guloso, ele garante que o destino seja encontrado sempre pelo menor caminho, não havendo necessidade de continuar verificando os demais vértices pertencentes a *Q*.

Outro trecho de código a ser destacado encontra-se na linha 7, pois nesta linha ocorre a escolha do vértice de menor valor de distância até a origem dentre os vértices que ainda não foram

visitados. O vértice escolhido é removido de *Q* e identificado como *u*, tornando-se o pivô do algoritmo, pois todas as demais interações e condições estarão relacionadas a ele.

Estando com *u* e este não sendo o vértice destino, inicia-se uma série de verificações a partir dele. Para todos os vértices adjacentes a *u* (linha 10), que ainda não foram averiguados (linha 11), afere-se se a taxa de perda de pacotes (linha 11) e a latência (linha 12) de *v* estão condizentes com as condições estabelecidas pelo administrador, apenas se as duas condições forem atendidas simultaneamente para cada vértice; em caso afirmativo este será inserido em *Q* (linha 13). Antes da inserção se *v* na lista *Q*, verifica-se se este já foi incluído na lista oriundo de outro caminho; caso esteja na lista, analisa-se se este possui valor de distância até a origem menor que a percorrida pelo caminho atual (linha 14); caso o valor de distância do caminho atual seja menor, ao invés de inseri-lo em *Q*, apenas altera-se seu valor de distância para o atual (linha 15) e troca-se o caminho até achegar ao vértice *v* (linha 16).

4.4 Implementação da Proposta

O Q-Path integra a necessidade de utilização do núcleo de um controlador OpenFlow, à medida que as funcionalidades básicas do OpenFlow estão em um nível de maturidade bastante consistentes nos controladores. O controlador escolhido para a implementação da proposta foi Floodlight[5]. A escolha desse controlador se baseou em algumas justificativas, tais como:

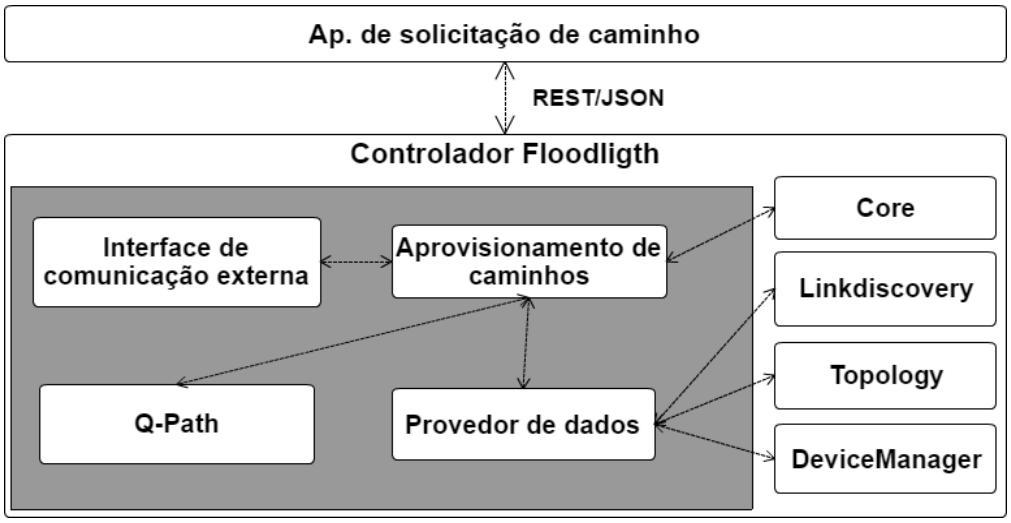
- Código *open source*: por ser um código aberto, podemos efetuar alterações livremente no controlador, inserindo os códigos de avaliação da proposta;
- Possuir código modularizado: desta forma facilita a inclusão e novas funcionalidades, necessitando apenas da ligação do novo módulo ao núcleo do controlador;
- A linguagem de programação: por ser baseado em Java. Como já possuímos conhecimento aprofundado da linguagem, foi possível a fácil compreensão do código.

Além desses critérios de escolha, esse era o controlador utilizado no laboratório de pesquisa, no período do desenvolvimento da proposta.

[5] http://www.projectfloodlight.org/

A Figura 10 apresenta a estrutura do controlador Floodlight em módulos de conexão com uma aplicação externa que viabiliza a interação do administrador da rede com o Q-path. A caixa cinza em evidência apresenta os módulos adicionados ao controlador, viabilizando assim a implementação da proposta. Além desses, foram destacados, dentre vários módulos do controlador, três módulos aos quais foi necessário maior interação para a coleta de informações bem como da vinculação dos módulos propostos ao controlador.

Figura 10 - Estrutura do controlador Floodlight em comunicação com aplicação externa.



Fonte: Elaborada pelo autor.

Além dos módulos já disponibilizados pelo controlador (*Core*, *Linkdiscovery*, *Topology*, *DiviceManager*), foram desenvolvidos mais 4 módulos necessários para o funcionamento da proposta Q-Path: (1) Interface de comunicação externa; (2) Aprovisionamento de caminhos; (3) Provedor de dados e (4) Q-Path. A seguir um detalhamento desses módulos.

1. **Interface de comunicação externa**: é responsável por interagir com a aplicação externa e receber os requisitos necessários para a busca do caminho desejado. As solicitações de busca ocorrem por meio de mensagens em formato de serviços web (*web services*). Além de realizar o controle de envio e recebimento de mensagens, outra importante funcionalidade deste módulo é realizar o *parser* de mensagens de acordo com o padrão definido para a estrutura da mensagem, efetuando uma separação entre cada elemento, enviando-os para o módulo de **aprovisionamento de caminhos,** de forma estruturada.
2. **Provedor de dados**: recolhe todas as informações necessárias para a busca, a partir da rede. Acessando módulos detentores de informações do controlador, como *topology*, *linkdiscovery* e *deviceManager*, coletam-se todos os dados necessários

para uma consulta, tais como descoberta da topologia com seus respectivos dados estatísticos de enlaces e dispositivos (por ex.: perda de pacotes, atraso entre os enlaces e qualquer outro dado estatístico que possa estar disponível). Os dados são atualizados sempre que houver alguma alteração na rede, seja inserção de novo *switch*, seja enlace, seja solicitação de nova rota. Em geral os controladores já disponibilizam módulos de monitoramento da rede. Além disso, o módulo armazena as informações a respeito dos elementos de comutação, dos enlaces e suas estatísticas coletadas, construindo, assim, um banco de dados de engenharia de tráfego (TED – *Traffic Engineering Database*), requisito necessário para configuração dos enlaces, mantendo uma visão topológica da rede. Os dados são armazenados em uma estrutura de grafos contendo todas as características que possam refletir a rede de forma real. Com essa estrutura de dados refletindo a topologia de rede será efetuada a busca, utilizando o algoritmo Q-Path.

3. **Q-Path:** é o componente responsável por computar o caminho, contendo o algoritmo de busca apresentado na seção 4.3. Tem a capacidade de encontrar o caminho, baseando-se em um valor de latência mínimo a ser percorrido pelo algoritmo e em um valor aceitável (relativo ao administrador da rede) de perda de pacotes por fila no roteador.
4. **Aprovisionamento de caminhos**: responsável pela interação com a interface, recebendo as informações repassadas pela aplicação e utilizando-as para solicitar o melhor caminho ao módulo Q-Path, passando como parâmetro a topologia atualizada (provida pelo módulo **Provedor de dados**), bem como as informações que condicionam o algoritmo. A busca é efetuada baseada nos requisitos passados pela aplicação. Como resultado, caso haja um caminho entre os nós, retorna ao solicitante (aplicação ou administrador da rede) um conjunto de nós (ordenados inversamente) referentes ao caminho encontrado; caso contrário, uma mensagem é enviada informando a inexistência do caminho.

A *Northbound Interface* é uma camada muito relevante para a arquitetura da SDN (Nunes *et al*, 2014), pois ela é uma API que faz a comunicação com o plano de controle efetivamente, permitindo que aplicações utilizadas por administradores de rede efetuem o controle e o monitoramento das funções da rede sem necessariamente ajustar os detalhes mais finos (lidar com linhas de comando) da comunicação.

As funções principais de uma interface *Northbound* são: traduzir os requisitos das aplicações de gerenciamento em instruções de baixo nível para os dispositivos da rede e transmitir estatísticas sobre a rede, que foram geradas nos dispositivos da rede e processadas pelo controlador.

Com o intuito de permitir ao administrador da rede acompanhar o estado da rede e efetuar buscas de caminhos, neste trabalho foi desenvolvida uma interface na ponte norte com o objetivo de permitir tal acompanhamento.

Utilizando *web services,* que é uma forma de comunicação que permite interoperabilidade, optou-se pelo protocolo REST por ser mais leve em relação ao protocolo comumente utilizado na comunidade, o SOAP. REST utiliza JSON para efetuar as requisições de intercâmbio; REST é uma sintaxe para armazenamento e troca de informações, de fácil manuseio e uma alternativa mais leve do que o XML utilizado pelo protocolo SOAP. A utilização de REST/JSON depende, em muitos casos, unicamente de uma URL (*Uniform Resource Locator*) simples para efetuar as requisições.

5 ANÁLISE DA PROPOSTA

Este capítulo objetivou validar a proposta do algoritmo Q-Path. Para atingir tal objetivo foi realizada a implementação desta proposta, conforme descrita no capítulo anterior. Para isso são realizados diversos testes relacionados ao algoritmo, executando em diversas condições e topologias diferentes para aferir métricas que atestem a eficácia e eficiência da proposta.

5.1 Ferramentas Utilizadas

As simulações foram realizadas para comparar o desempenho do algoritmo proposto, o Q-Path, com o algoritmo comumente empregado nos controladores atuais, o Dijkstra. Dentre os controladores disponíveis foi utilizado o Floodlight versão 0.91, que é uma implementação de controlador OpenFlow baseado em linguagem Java, com licença Apache, desenvolvido e mantido pela comunidade livre, incluindo engenheiros do *Big Switch Networks*[6]. Adicionalmente, é um controlador bem estruturado e modularizado, facilitando a diferenciação entre os módulos propostos no trabalho. Haja vista este trabalho primar pela inclusão de módulos no controlador, esses foram desenvolvidos utilizando a linguagem de programação Java com JDK (*Java Development Kit*) versão 7.

O ambiente utilizado para a criação de topologias para a realização dos testes foi o Mininet[7]. Trata-se de um emulador de redes desenvolvido por pesquisadores da Universidade de Stanford, Estados Unidos, com o objetivo de apoiar pesquisas colaborativas, permitindo protótipos autônomos de SDN, a fim de que qualquer pessoa possa fazer o download, executar, avaliar, explorar e ajustar.

A coleta dos dados foi efetuada em uma máquina com processador Intel® Core™ i5-3330@ 3,00GHz, Memória 8Gb DDR3, no qual tanto o controlador Floodlight quanto a máquina virtual (instalada via Virtual Box[8]), com o mininet instalado, estavam rodando. As solicitações da busca foram passadas via navegador (descritos na seção 4.4), por meio de uma interface desenvolvida para facilitar a interação do administrador da rede com a aplicação. Além dessas ferramentas,

[6] http://www.bigswitch.com/

[7] Disponível em: <http://mininet.org/>.

[8] https://www.virtualbox.org/

utilizou-se o JMeter[9] (uma ferramenta utilizada para testes de carga em serviços oferecidos por sistemas computacionais e assim como o controlador, também é um projeto da *Apache Software Foundation*[10]) para a realização das mil solicitações de caminho.

5.2 Metodologia dos Experimentos

Para viabilizar as pesquisas na área de redes, existem algumas alternativas como modelagem analítica, implementação de protótipo e simulação. Em virtude da complexidade associada à implantação de um protótipo real de tamanha magnitude para SDN é interessante, em um primeiro momento, a realização de estudos de ambientes computacionais que permitam a simulação de diferentes cenários, bem como o teste de soluções propostas dentro do contexto apresentado.

A fim de validar a proposta de algoritmo sugerida no contexto deste trabalho, foi escolhida a opção de simular topologias reais por meio de um *software* capaz de reproduzir uma rede SDN de forma virtual, o Mininet (LANTZ; HELLER; MCKEOWN, 2010).

As topologias escolhidas para os experimentos foram Rede IPÊ, Internet2 e GEANTs. Estas foram escolhidas por serem exemplos de topologias reais que possuem uma grande variedade de nós e enlaces, tornando cada uma delas ideal para as aplicações de testes do algoritmo. Assim, possibilitam avaliar melhor o comportamento do algoritmo.

As topologias foram criadas no mininet, utilizando da linguagem Python. Por não possuirmos os valores aproximados de atraso de cada enlace nem de perda de pacotes de cada roteador e considerando que alguns desses valores são variáveis, além disso, objetivando não comprometer os experimentos, optou-se pela geração aleatória desses valores. Para cada atraso e perda de pacotes, gerava-se valores entre 1 e 10. Assim, para cada simulação realizada, os valores de enlaces e perda de pacotes estariam variando, forçando os algoritmos a encontrarem sempre caminhos diferentes, mesmo que recebessem os mesmo parâmetros de entrada.

Para uma análise do algoritmo de Dijkstra, eram necessários como parâmetro de entrada, apenas os dados de origem e destino, enquanto que para Q-Path eram passados, além destes, valores de limite para a latência e limite de perda de pacotes. Como demonstrado nas tabelas

[9] http://jmeter.apache.org/

[10] https://www.apache.org/

(apresentadas na seção 5.3), para cada origem e destino há um valor de alcance estimado diferenciado, que pode ser coletado a partir de estimativas ou uma possível tabela de histórico. Para a perda de pacotes, como sugerido pela Resolução nº 574 da Anatel, valor máximo de 2%, optou-se por manter esse valor fixo em todos os experimentos, considerando a geração para esses valores variando de 1 a 10%.

Além dessas topologias, a fim de avaliar aspectos de escalabilidade da solução, foi desenvolvido uma geração de topologias em malha variando de 16 a 100 nós. Esta geração criava uma topologia em malha de acordo com o grau desejado para o grafo, grau 4 equivale a 16 nós, grau 5 a 25 nós e assim sucessivamente até grau 10 com 100 nós.

Para a aferição dos dados, os mesmos testes foram realizados em iguais condições para os algoritmos avaliados. A coleta de informações seguiu os seguintes passos:

- Após a topologia ser reconhecida pelo controlador, foram escolhidos, dentre os nós da rede, 20 origens e destinos diferentes e, para cada um dos pares (origem/destino), foram efetuadas 70 solicitações de caminhos, totalizando 1400 buscas para cada topologia;
- Para cada busca efetuada, o controlador realizava uma escrita em um arquivo de texto, registrando o tempo de execução de cada solicitação de caminho;
- Após a execução das 1400 solicitações de caminhos, retirou-se uma amostra de 1000 registros de cada topologia para o cálculo da média aritmética;
- Para cada gráfico apresentado foram utilizadas barras de erros com intervalo de 5% em relação a media dos valores;
- Criação dos gráficos a partir dos resultados.

5.3 Cenários de Avaliação

Como citado anteriormente, foram utilizadas três exemplos de topologias reais. A primeira é uma topologia baseada na rede nacional de pesquisa (Rede Ipê[11]); a segunda baseou-se no *backbone* da Internet2[12] nível 2; e a terceira foi baseada na rede europeia GEANTS[13].

[11] https://www.rnp.br/

[12] http://www.internet2.edu/

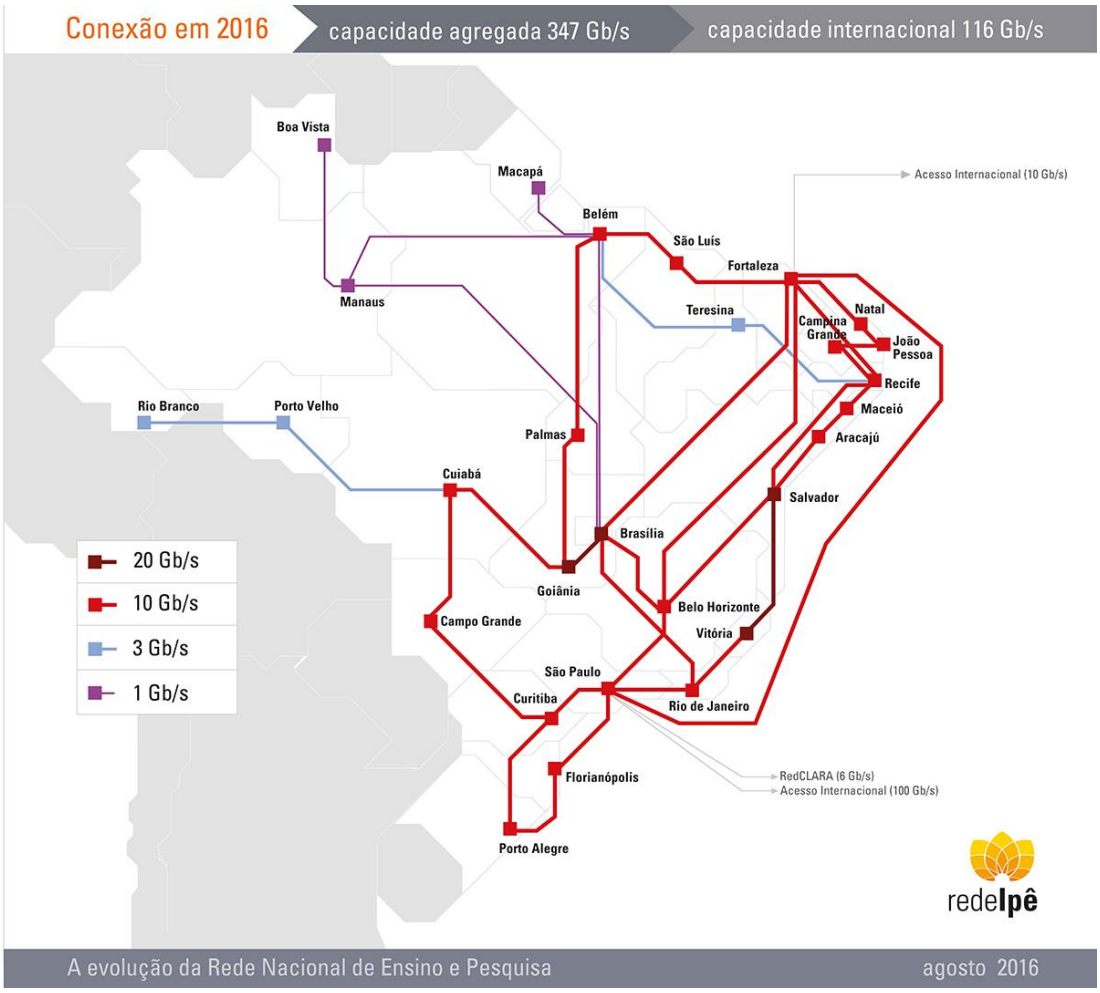
[13] http://www.geant.org/

Além dessas topologias, também foram avaliados aspectos de escalabilidade do algoritmo, utilizando topologias em malha variando de 16 a 100 nós de rede, possibilitando avaliar a eficiência do algoritmo em aspectos diferenciados de uma topologia de rede comum.

5.3.1 Rede Ipê

A Figura 11 apresenta a Rede IPÊ, operada pela RNP (Rede Nacional de Pesquisa). Essa é uma infraestrutura de rede de internet dedicada à comunidade brasileira de ensino superior e pesquisa, que interconecta universidades e seus hospitais, institutos de pesquisa e instituições culturais.

Figura 11 - Topologia Rede Ipê.



Fonte: Extraído de Rede Ipê[14].

[14] https://www.rnp.br/servicos/conectividade/rede-ipe

Inaugurada em 2005, foi a primeira rede óptica nacional acadêmica a entrar em operação na América Latina. Seu *backbone* foi projetado para garantir não somente a velocidade necessária ao tráfego de internet de aplicações básicas (navegação web, correio eletrônico e transferência de arquivos), mas também ao tráfego de serviços, aplicações avançadas, projetos científicos, e à experimentação de novas tecnologias, serviços e aplicações.

Atualmente, a infraestrutura da rede Ipê engloba 27 Pontos de Presença (PoPs), um em cada unidade da federação, além de ramificações para atender 1219 campi e unidades de instituições de ensino, pesquisa e saúde em todo o país, beneficiando mais de 3,5 milhões de usuários (Rede Ipê, 2016).

Na Tabela 3 encontram-se os dados de entrada para cada solicitação de caminho para os dois algoritmos, sendo estes, origem, destino e o atraso estimado. Este último parâmetro é utilizado apenas por Q-path.

Tabela 3 - Dados de entrada para algoritmos na Rede Ipê.

Nº	Origem	Destino	Atraso estimado	Nº	Origem	Destino	Atraso estimado
1	São Paulo	São Luís	25 ms	**11**	Macapá	Florianópolis	45 ms
2	Fortaleza	Rio de Janeiro	30 ms	**12**	Fortaleza	Porto Alegre	25 ms
3	Maceió	Rio Branco	70 ms	**13**	Rio de Janeiro	Recife	30 ms
4	Boa Vista	Fortaleza	30 ms	**14**	Campo Grande	Natal	30 ms
5	Boa Vista	Rio de Janeiro	50 ms	**15**	Manaus	Cuiabá	35 ms
6	Boa Vista	Porto Alegre	50 ms	**16**	Boa Vista	Rio Branco	50 ms
7	Rio Branco	João Pessoa	70 ms	**17**	Macapá	Rio Branco	45 ms
8	Salvador	Belém	35 ms	**18**	São Luis	Vitória	40 ms
9	Goiânia	Porto Alegre	35 ms	**19**	Belém	Rio Branco	40 ms
10	Brasília	Aracajú	30 ms	**20**	Belém	Porto Alegre	40 ms

Fonte: Elaborada pelo autor.

5.3.2 Internet2

A Internet2, apresentada na Figura 12, é uma iniciativa norte-americana voltada para o desenvolvimento de tecnologias e aplicações avançadas, fundada pelas principais instituições de ensino superior do país em 1996. A internet2 provê um ambiente colaborativo, no qual as organizações educacionais e de pesquisa norte-americanas podem resolver desafios tecnológicos comuns e desenvolver soluções inovadoras em apoio às missões de serviço à comunidade, pesquisa e educação.

Figura 12 - Topologia Internet2.

Fonte: Extraído de Internet2[15].

Internet2 serve a mais de 90.000 instituições comunitárias âncoras, 305 universidades norte-americanas, 70 agências governamentais, 42 redes de ensino regionais e estaduais, 84 principais empresas que trabalham com a nossa comunidade e mais de 65 parceiros de pesquisa e redes de educação nacionais que representam mais de 100 países (INTERNET2, 2016).

Na Tabela 4 encontram-se os devidos dados de entrada utilizados na realização dos testes nessa rede, similarmente a rede Ipê.

---

[15] https://www.internet2.edu/media/medialibrary/2016/10/06/I2-Network-Infrastructure-Topology-Layer2-201610.pdf

Tabela 4 - Dados de entrada para algoritmos na Rede Internet2.

Nº	Origem	Destino	Atraso estimado	Nº	Origem	Destino	Atraso estimado
1	Seattle	Baton Rouge	60 ms	**11**	Portland	Albany	80 ms
2	Portland	Indianapolis	50 ms	**12**	Salt Lake City	Baton Rouge	55 ms
3	Sunnyvale	Louisville	70 ms	**13**	Boston	Seattle	80 ms
4	Kansas City	Salt Lake City	30 ms	**14**	Missoula	Washington DC	80 ms
5	El Paso	Columbia	50 ms	**15**	Jacksonville	Salt Lake City	60 ms
6	Reno	Minneapolis	45 ms	**16**	Raleigh	Sunnyvale	70 ms
7	Los Angeles	Jackson	50 ms	**17**	Los Angeles	Cleveland	70 ms
8	Indianapolis	Salt Lake City	45 ms	**18**	New York	Denver	70 ms
9	Dallas	Las Vegas	50 ms	**19**	Kansas City	Hartford	60 ms
10	Kansas City	Missoula	30 ms	**20**	Seattle	New York	80 ms

Fonte: Elaborada pelo autor.

5.3.3 GEANT

O projeto GÉANT (Rede Gigabit de pesquisa pan-européia), apresentado na Figura 13, começou em Novembro de 2000, tornando-se totalmente operacional em Dezembro de 2001. É uma rede de alta capacidade que engloba mais de três mil instituições de ensino e pesquisa em 32 países, através de 28 redes nacionais e regionais de ensino e pesquisa. Atuando desde dezembro de 2001, seu objetivo principal é dar continuidade e melhorar a versão anterior da rede de pesquisa pan-européia, TEN-155[16].

A rede GÉANT é a principal da Europa, direcionada à pesquisa e educação por meio através de ligações na ordem dos *Gigabits*. As ligações de GÉANT variam entre os 1 Gbps, por meio de equipamentos mais antigos, até 100 Gbit/s por ligações de fibra óptica.

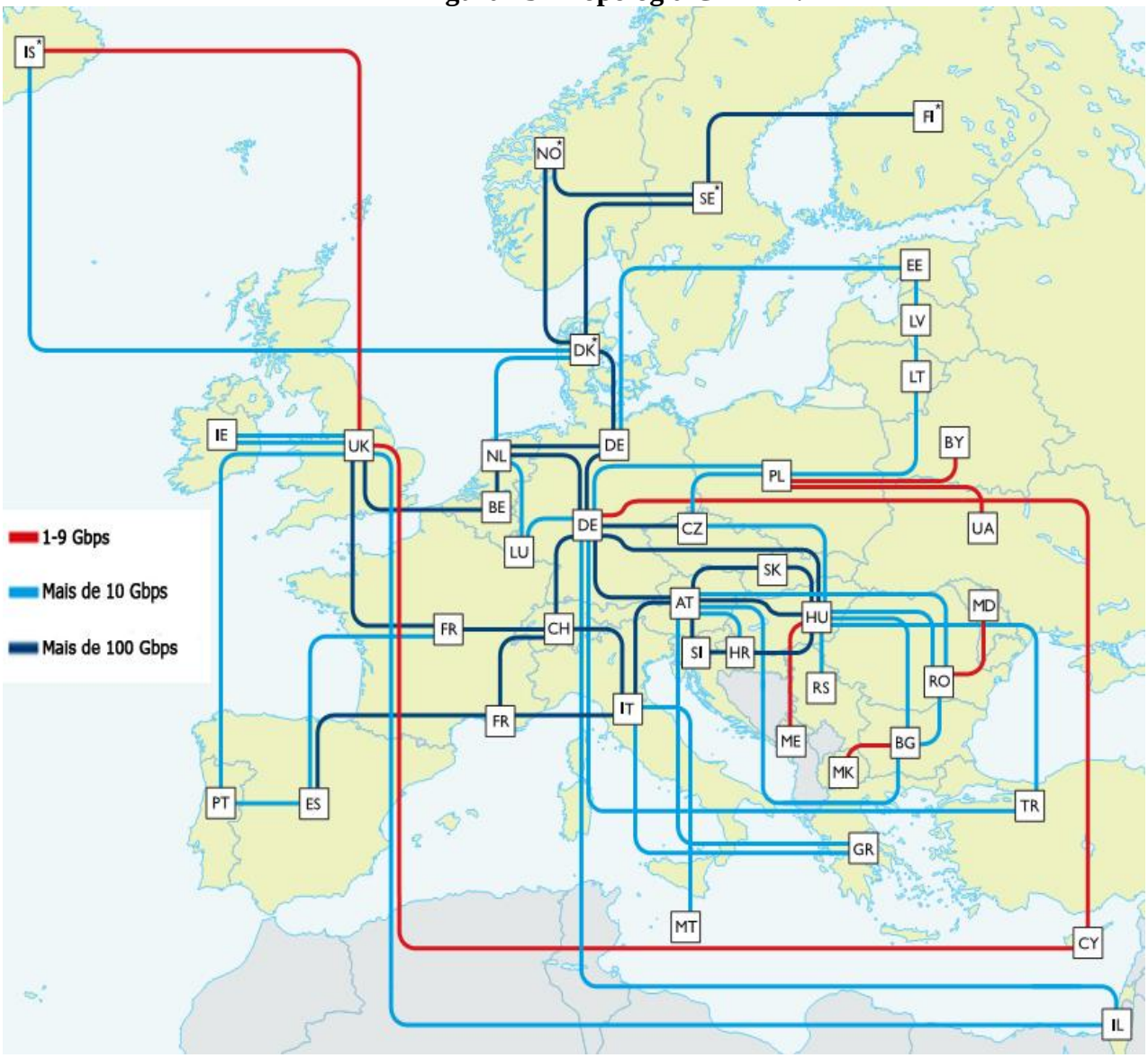
Essa rede oferece uma tecnologia que os operadores comerciais não são capazes de oferecer. Uma rede de alta velocidade que ultrapassa os limites da tecnologia da rede, proporcionando, simultaneamente, uma infraestrutura pan-européia de baixo custo. Grandes projetos de pesquisa contam com GÉANT para uma excelente disponibilidade e qualidade do serviço (GÉANT, 2015).

A rede GÉANT oferece uma internet de ultra velocidade disponível apenas para educação e pesquisa. Totalizando, atualmente, mais de 1.000 terabytes de dados transferidos por dia através

[16] http://www.gateway.nameflow.net/ten-155/

do backbone IP. A capacidade de transmissão chega a 100 Gbps (gigabits por segundo), porém seu projeto de rede original é suportar até 8Tbps (terabits por segundo), assegurando que a rede permaneça à frente da demanda do usuário e do dilúvio de dados (GÉANT, 2015).

Figura 13 - Topologia GÉANT.



Fonte: Extraído de GÉANT[17].

Os testes na rede GÉANTs foram realizados a partir nos dados de entrada apresentados na tabela 5.

[17] http://www.geant.org/Resources/Documents/topology_map-16OCT15.PDF

Tabela 5 - Dados de entrada para algoritmos na Rede GÉANT.

Nº	Origem	Destino	Atraso estimado	Nº	Origem	Destino	Atraso estimado
1	IS	FI	40 ms	**11**	BG	UK	30 ms
2	IS	TR	40 ms	**12**	HR	IS	30 ms
3	EE	GR	40 ms	**13**	IL	FI	45 ms
4	RO	IL	30 ms	**14**	PT	BY	55 ms
5	DK	MT	40 ms	**15**	CY	NO	45 ms
6	BY	PT	45 ms	**16**	ES	DK	40 ms
7	NO	ES	50 ms	**17**	PL	SE	40 ms
8	LV	CH	45 ms	**18**	FR	LV	50 ms
9	CY	UK	15 ms	**19**	MK	UK	40 ms
10	FI	FR	40 ms	**20**	DE	MT	35 ms

Fonte: Elaborada pelo autor.

5.4 DISCUSSÃO DOS RESULTADOS OBTIDOS

Esta subseção apresenta os resultados obtidos nas simulações, cujo fator principal a ser analisado foi referente ao tempo de busca de caminho para cada uma das topologias utilizadas. Além deste parâmetro, foram apresentados gráficos referentes ao consumo de memória e processamento do controlador pela aplicação. O objetivo é avaliar o impacto do mecanismo de seleção de rotas proposto na obtenção de garantias de qualidade de serviço em redes SDN.

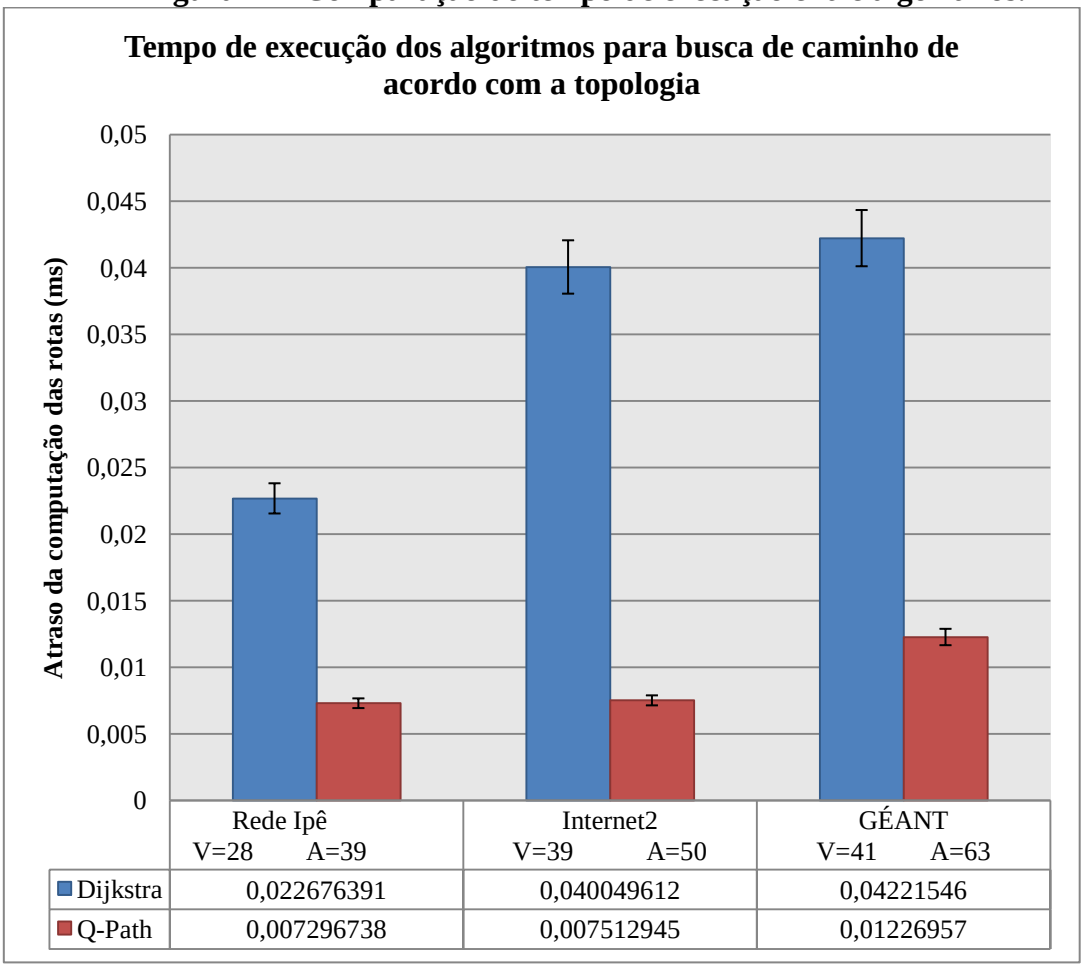
Todas as simulações topológicas foram conduzidas com 1400 repetições para cada topologia e utilizando uma amostra de 1000 registros de cada para análise dos dados. A escolha de um valor de amostragem foi necessária porque nem todos os parâmetros passados ao Q-Path garantiam um caminho, ou seja, para algumas das solicitação de caminhos o controlador respondia que não existia caminho; para esses casos os valores não foram considerados para análise dos dados a fim de não comprometer os resultados, sendo utilizados apenas os valores de tempo em que Q-Path encontrava um caminho para os dados de entrada.

5.4.1 Comparação entre as topologias

O primeiro resultado gerado trata-se da coleta do tempo médio necessário para a busca de caminhos dentro das topologias espelhadas nos exemplos reais. A Figura 14 dispõe os resultados dessas buscas. Analisando o gráfico apresentado na Figura 14, nota-se a vantagem do Q-Path em relação ao algoritmo de Dijkstra, pois para a topologia da rede Ipê, constata-se que o tempo médio de busca de caminho do algoritmo Q-Path, corresponde a 32,18% do tempo gasto em Dijkstra, ou seja, enquanto o tempo médio do Q-Path foi de aproximadamente 0,007 ms, para Dijkstra foi necessário cerca de 0,022 ms. Paralelamente, a rede GÉANT demonstrou aumento nessa

diferença, pois Q-Path condiz com apenas a 29,06% do tempo gasto na execução do algoritmo de Dijkstra. A desigualdade é mais evidente na rede Internet2, na qual Q-Path corresponde a 18,76% do tempo de Dijkstra.

Figura 14 - Comparação do tempo de execução entre algoritmos.



	Rede Ipê V=28 A=39	Internet2 V=39 A=50	GÉANT V=41 A=63
Dijkstra	0,022676391	0,040049612	0,04221546
Q-Path	0,007296738	0,007512945	0,01226957

Fonte: Elaborada pelo autor.

Efetuando uma análise sobre a estrutura fisiológica das redes para algoritmo de Dijkstra, percebe-se que o algoritmo aumentou seu tempo de busca após o aumento considerável do número de vértices, por exemplo, partindo da rede Ipê para Internet2, com um aumento de 39,28% no número de vértices (28 vértices na rede Ipê para 39 na Internet2) o tempo de procura do caminho aumentou em 76,61%. Comparando-se a rede Ineternet2 e a rede GÉANT, nota-se que com aumento de 5,12% no número de vértices houve um aumento de 5,41% no tempo de busca. Desta forma podemos concluir uma forte relação no algoritmo de Dijkstra com o aumento do número de vértices.

Observando Q-Path em relação às topologias, verificamos que com o mesmo aumento, de 39,28%, no número de vértices, Q-Path teve um aumento de apenas 2,96% no tempo de busca de

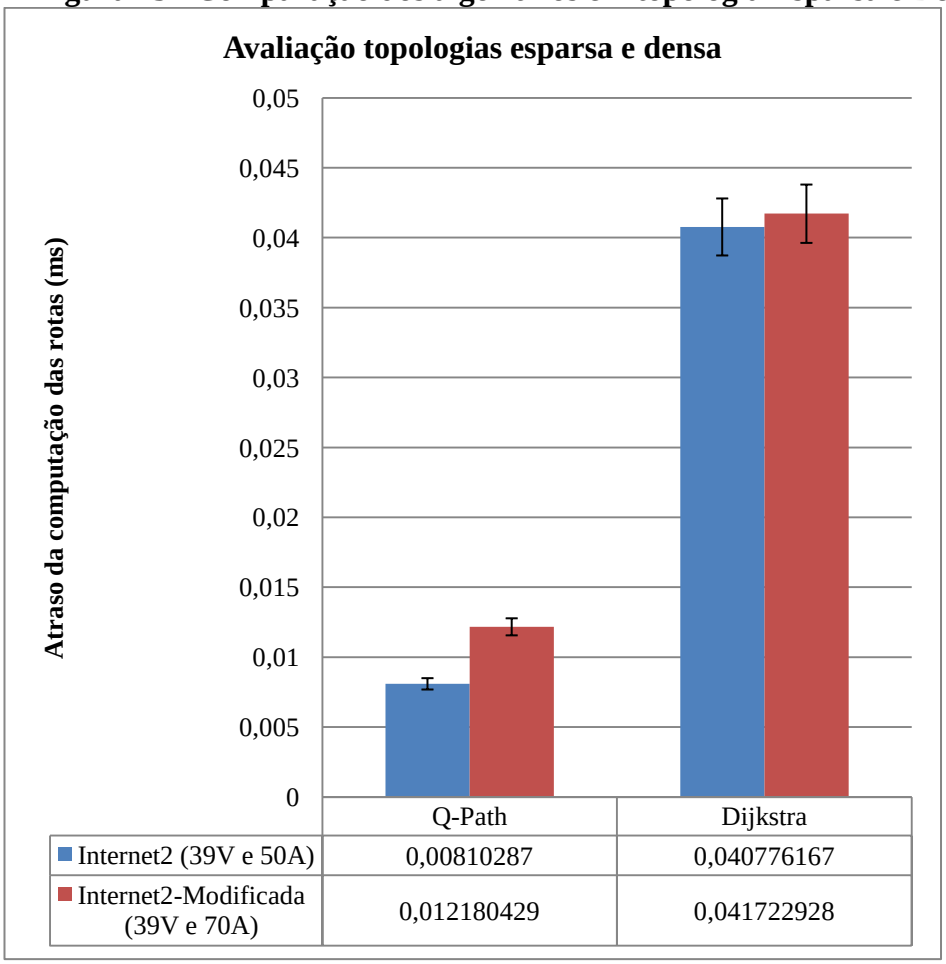
caminhos. Em contra partida, com o aumento de 5,12% da rede Internet2 em relação a GÉANT, o tempo de busca aumentou em 63,31%, permitindo a inferência de ausência de relação com o número de vértices do grafo. No entanto, ao analisar o número de arestas, pode-se constatar que houve um aumento de 28,20% da rede Ipê em relação à Internet2, ou seja, inferior ao aumento do número de vértices. Por outro lado, apesar de um aumento de 5,12% do número de vértices, da topologia Internet em relação à GÉANT, o número de aresta aumentou em 26%. Com isso concluímos que Q-Path é mais eficiente em grafos esparsos do que em grafos densos - Pozzer (2008) define como grafos esparsos os que possuem poucas conexões por nó e grafos densos os que possuem muitas conexões por nó.

Objetivando uma maior análise quanto à influência da densidade da topologia para os algoritmos, obtivemos os resultados apresentados na Figura 15, os quais ilustram a topologia Internet2 com um aumento de vinte vértices, ou seja, a topologia modificada passou a conter 70 arestas. Constatou-se um aumento pronunciado em Q-Path, quanto aos resultados da modificação da topologia, pois, ao incrementar vinte vértices na topologia o tempo médio de busca aumentou em 50,32% em relação a topologia Internet2 real. Em contra partida, o algoritmo de Dijkstra teve um aumento de 2,32%. Assim, observou-se que Q-Path é mais eficiente em relação ao tempo para grafos esparsos, ressaltando-se que as topologias de rede em geral se apresentam mais semelhantes a grafos esparsos, reforçando a importância dos dados obtidos. Adicionalmente, observou-se que apesar do aumento de 50,32%, Q-Path ainda é consideravelmente mais eficiente que Dijkstra.

Outro fator relevante quanto ao aumento da densidade de uma topologia é que apesar do algoritmo proposto demandar um pouco mais de tempo, aumenta também a capacidade de encontrar uma rota existente. Analisando a quantidade de caminho encontrada, podemos constatar que o algoritmo de Dikjstra encontra 100% das rotas nas duas topologias, enquanto que Q-Path encontrou para a topologia esparsa aproximadamente 78% dos caminhos dentro dos parâmetros passados. Em contrapartida, ao aumentar a quantidade de arestas, aumentou-se também a quantidade de novas possibilidades de caminhos e Q-Path passou a encontrar aproximadamente 94% dos caminhos solicitados.

A curvatura apresentada no gráfico da Figura 16 comprova a relação de complexidade apresentada na seção 3.3, em que Q-path, mesmo possuindo um crescimento exponencial, assim como Dijkstra, apresenta um crescimento inferior.

Figura 15 - Comparação dos algoritmos em topologia Esparsa e Densa.

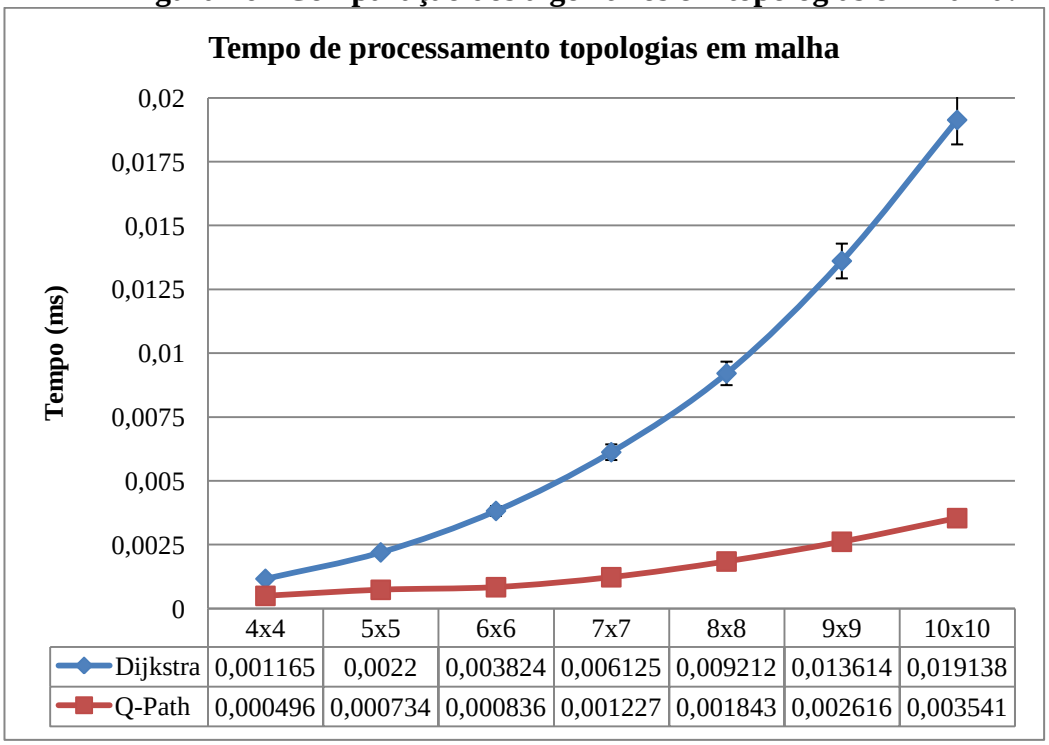


Fonte: Elaborada pelo autor.

5.4.2 Topologias em malha

A análise da Figura 16, gerada a partir das topologias em malha (grafos que crescem proporcionalmente em números de vértices e de arestas), constatou-se que os dois algoritmos geraram curvas exponenciais, porém com curvaturas diferentes. Para a topologia 4x4 (16 vértices, 24 arestas), Q-Path corresponde a 41,67% do tempo necessário em Dijkstra; para a topologia 5x5 (25 vértices, 40 arestas), Q-Path corresponde a 31,81% e, notoriamente, esta diferença aumenta conforme o aumento do tamanho das topologias. Q-Path chega a equivaler a 18,32% do tempo utilizado em Dijkstra para a topologia 10x10 (100 vértices, 180 arestas).

Figura 16 - Comparação dos algoritmos em topologias em malha.



	4x4	5x5	6x6	7x7	8x8	9x9	10x10
Dijkstra	0,001165	0,0022	0,003824	0,006125	0,009212	0,013614	0,019138
Q-Path	0,000496	0,000734	0,000836	0,001227	0,001843	0,002616	0,003541

Fonte: Elaborada pelo autor.

5.4.3 Consumo de CPU de Memória

Para avaliar o consumo de processamento e memória foi utilizado a JvisualVM[18], uma ferramenta desenvolvida pela Oracle, útil para o monitoramento de processos que rodam na JVM (*Java Virtual Machine*). Haja vista o controlador Floodlight ser desenvolvido em Java, a JvisualVM é ideal para monitorar o cosumo de processamento e memória de uma processo específico desenvolvido em Java, ou seja, com essa ferramenta é possível avaliar diretamente cada algoritmo com o mínimo de interferência de outros processos do sistema operacional.

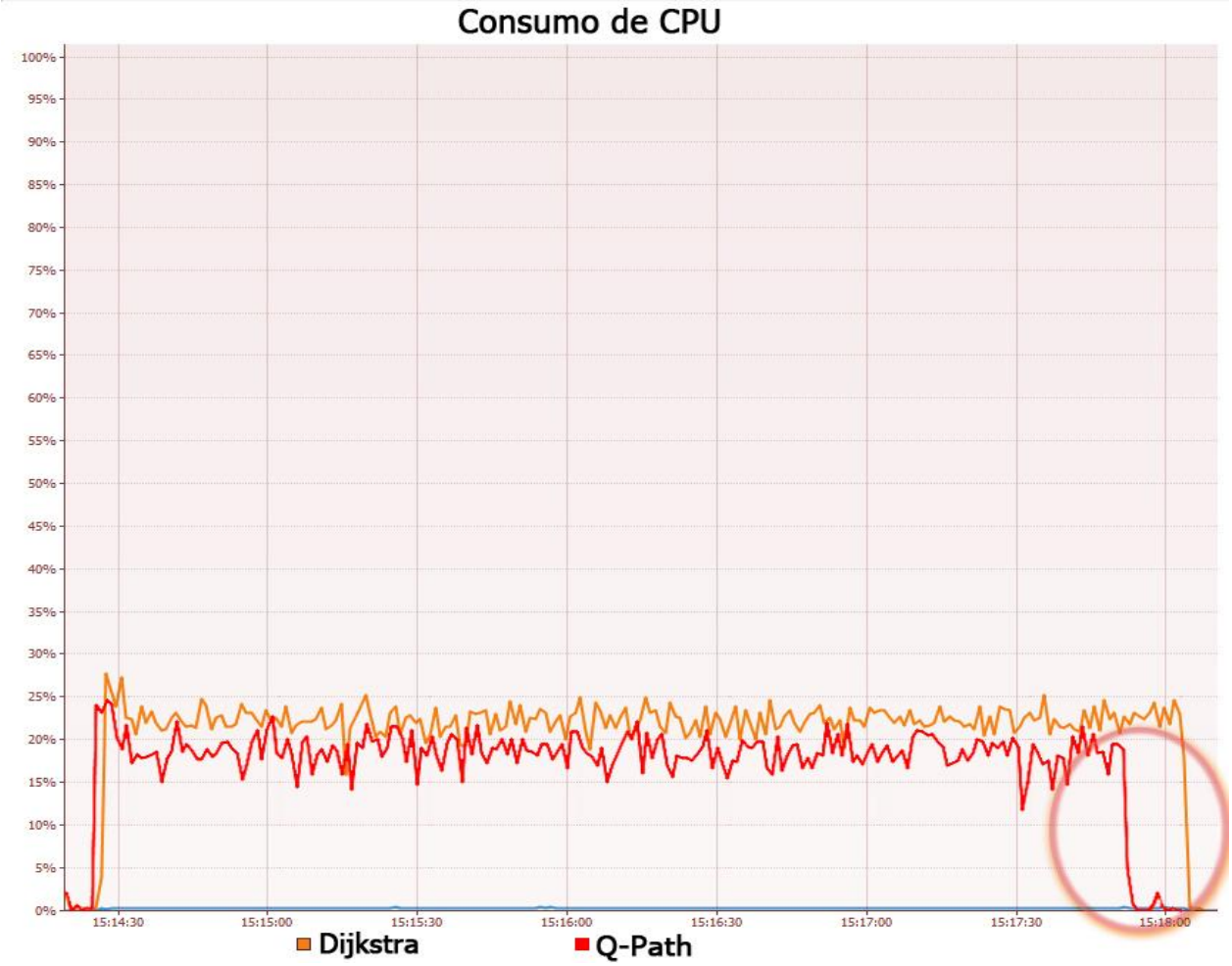
Para a coleta dos gráficos apresentados nas Figuras 17 e 18, com propósito de tornar mais evidente os gráficos, a carga de processamento foi maior. Utilizou-se a topologia em malha de tamanho 100x100, efetuando uma carga de 75.000 (setenta e cinco mil) solicitações de caminhos para as duas topologias, nas mesmas condições, assim haveria cosumo de processamento por mais tempo, aumentando a precisão dos dados.

[18] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jvisualvm.html

No gráfico de consumo de processamento (Figura 17), nota-se uma pequena diferença de consumo entre os algoritmos, observando-se mais uma vez Q-Path se apresentando de maneira mais eficiente que o algoritmo de Dijkstra. Verificando-se os dados obtidos e ilustrados na Figura 17, os dois algoritmos iniciam a execução com um pico de processamento, os dois algoritmos chegam a valores próximos, com uma diferença de aproximadamente 2% de consumo. Durante todo o restante da execução, Q-Path se mantém com processamento um pouco abaixo do processamento de Dijkstra. Destaca-se, neste gráfico, o término do processamento, pois Dijkstra consome mais CPU (*Central Processing Unit*) do que Q-Path, por ser discretamente mais demorado em seu tempo de busca.

No que se refere ao consumo de memória, apresentado na Figura 18, é possível inferir que para os mesmos valores de entrada para os gráficos, e mesmo que Q-Path possua um consumo de CPU discretamente menor, não houve aumento no consumo de memória em relação ao algoritmo de Dijkstra.

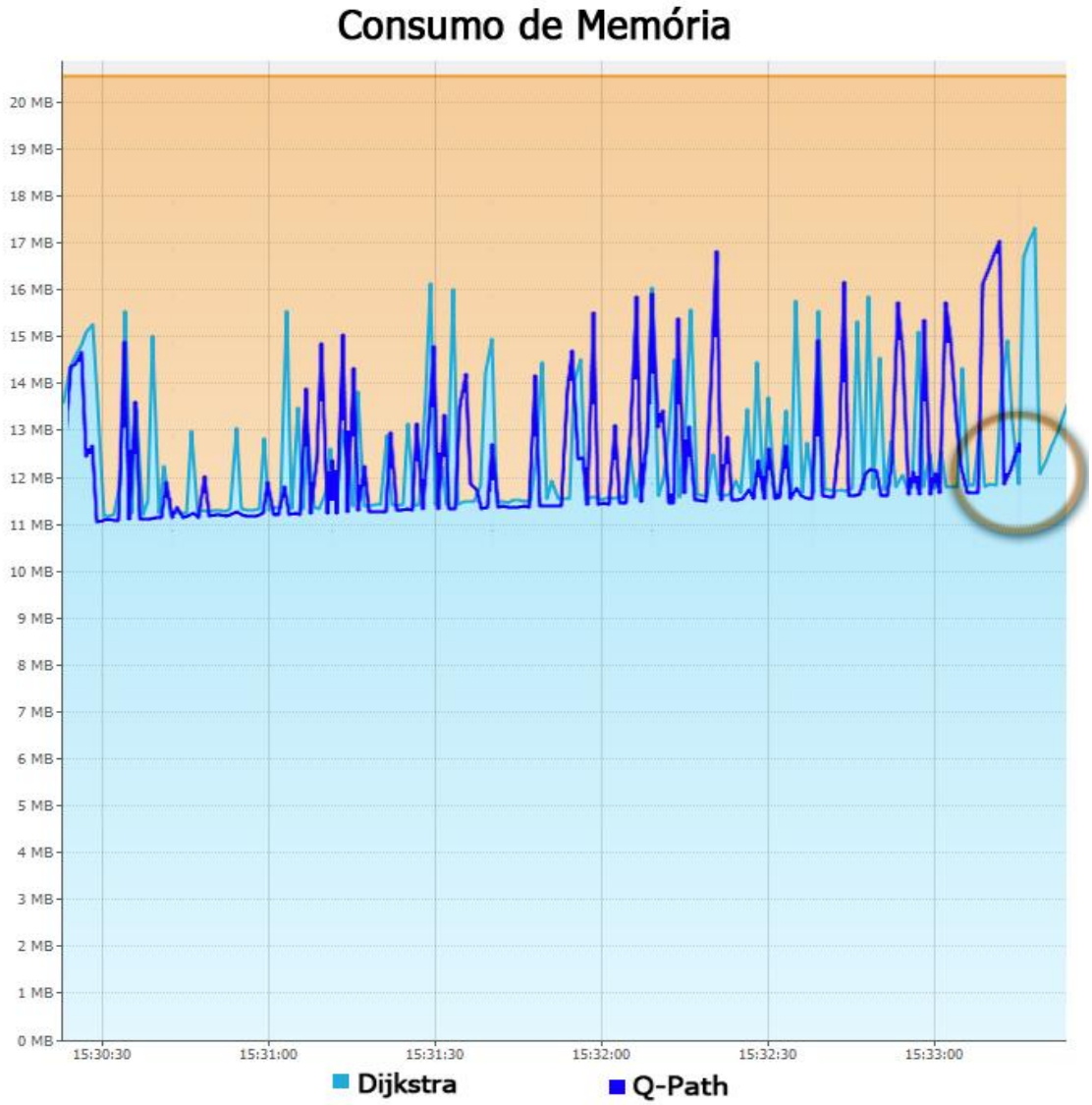
Figura 17 - Gráfico de análise de cosumo de processamento.



Fonte: Elaborada pelo autor.

Observa-se no gráfico da Figura 18 que em alguns momentos os cosumos de memória se intercalam entre os algoritmos; há momentos em que Q-Path consome menos memória, principalmente no início do processo (a esquerda do gráfico), porém há momentos em que esse mesmo algoritmo consome um pouco mais, chegando a ter um pico de consumo superior ao de Dijkstra. No entanto, apesar da proximidade no consumo de memória entre os algoritmos, Q-Path possui um consumo superficialmente menor ao analisar a parte inferior das ondulações. Além disso, Q-Path possui um consumo modestamente menor ao final do processo, uma vez que o algoritmo proposto possui um término de processamento menor do que Dijkstra (Figura 17), e assim a memória também passa a ser liberada um pouco antes do algoritmo de Dijkstra.

Figura 18 - Gráfico de análise de consumo de memória.



Fonte: Elaborada pelo autor.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esta dissertação apresentou um estudo sobre o modelo SDN/OpenFlow, descrevendo sua estrutura e principais características. Resumidamente, SDN é um modelo de rede de computadores que visa a separação dos planos de dados e controle nos comutadores, permitindo, assim, que o controle desses comutadores possa ocorrer por meio de aplicações externas e centralizadas em controladores SDN.

No entanto, para garantir QoS aos usuários, SDN necessita de mecanismos de cálculo de caminho que efetuem buscas com eficiência e eficácia, por exemplo, oferecendo cálculo otimizado de caminho dentro das especificações de QoS definidas por um administrador da rede ou de necessidades de uma aplicação. Neste contexto, foi elaborada a proposta Q-Path.

A descrição do algoritmo, a demonstração em pseudocódigo e um exemplo de implementação do Q-Path foram apresentados nesta pesquisa. Os detalhes apresentados na proposta fundamentaram os objetivos desta dissertação que foi propor um algoritmo otimizado de cálculo de caminho, utilizando parâmetros de QoS, em SDN.

Para a execução dos experimentos da dissertação, alguns exemplos de topologias foram criados no simulador de rede mininet, haja vista a inviabilidade da implementação de tal topologia em ambiente real. A implementação da proposta foi feita no controlador Floodlight. Durante as simulações, algumas métricas foram coletadas, como consumo de memória e processamento, com um destaque principal para o tempo de busca para encontrar o caminho solicitado.

Os resultados foram satisfatórios, sendo que o algoritmo proposto (Q-Path) produziu como resultados, por exemplo, medidas de performance como consumo de CPU durante o processamento inferior, mesmo com consumo de memória superficialmente equivalente ao algoritmo consagrado na literatura, concluindo o processamento e liberação de memória em menos tempo. O maior destaque da proposta foi o encontro do caminho solicitado com tempo consideravelmente menor que o algoritmo de Dijkstra.

Assim, por meio da apresentação do Q-Path, pretendeu-se contribuir para o avanço de pesquisas na área SDN. A realização das avaliações de resultados da implementação do algoritmo ocorreu por meio da execução em ambiente de simulação, porém isso não é impeditivo para a implementação do Q-Path em ambientes reais. O algoritmo encontra-se pronto para ser implementado em controladores que gerenciam redes reais.

6.1 Trabalhos futuros

Ainda que os resultados apresentados nesta dissertação sejam promissores, a proposta abre espaço para novas ideias e possibilidades de melhorias, como estas apresentadas a seguir.

Novas ideias:

- Efetuar testes com outros parâmetros de QoS, por exemplo, com custo do enlace, largura de banda e jitter, podendo inclusive incluir mais camadas de testes ou trocar por algum dos parâmetros utilizados. A utilização de outros parâmetros de QoS podem ser úteis, dependendo de para qual aplicação o algoritmo vai fazer a busca de caminho, uma vez que o QoS de algumas aplicações estão ligados a parâmetros específicos de QoS. É importante ressaltar que quanto maior o número de condições maior será a possibilidade de não encontrar o caminho. Em contrapartida, sabendo-se que o caminho almejado está dentro das condições impostas, há um aumento no desempenho do algoritmo, retornando ao caminho desejado rapidamente.

Possibilidades de melhorias:

- Primeiramente, para a implementação feita neste trabalho, há a necessidade de criação de uma interface externa mais amigável para o administrador, pois trabalhar *layout* estava fora do escopo desta dissertação.
- Outra melhoria que seria consideravelmente expressiva ao algoritmo seria uma criação, na origem do fluxo, de uma tabela com as informações estimadas para a latência. Isso facilitaria ao administrador da rede induzir valores mais próximos do real, por exemplo, para uma aplicação que exige latência máxima de 80 ms o administrador da rede não precisaria necessariamente incluir os 80 ms como valor limite para Q-Path, caso existisse uma tabela com conhecimento prévio de que a média de tempo gasto para encontrar aquele destino específico fosse de apenas 40 ms. Com uma pequena base de conhecimento na origem, aumentar-se-ia a precisão das informações passadas para Q-Path e este encontraria o caminho com mais rapidez.

7 BIBLIOGRAFIA

ARAÚJO, A. **As pontes de Königsberg**. Disponível em: <http://www.mat.uc.pt/~alma/escolas/pontes/>. Acesso em: 21 dez. 2016.

BARROS, E. A. R., PAMBOUKIAN, S. V. D. & ZAMBONI, L. C. **Algoritmo de Dijkastra: apoio didático e multidisciplinar na implementação, simulação e utilização computacional.** International Conference on Engineering and Computer Education, p. 960–963, 2007. Disponível em: <http://meusite.mackenzie.br/edsonbarros/publicacoes/ICECE2007_212.pdf>. Acesso em: 21 dez. 2016.

BHATTACHARYA, B.; DAS, D. **SDN based architecture for QoS enabled services across networks with dynamic service level agreement**. In: 2013 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), Anais IEEE, Dez. 2013. Disponível em: <http://ieeexplore.ieee.org/document/6802868/>. Acesso em: 20 dez. 2016.

BRADENAND, R.; ZHANG, L.; BERSON, S.; HERZOG, S.; JAMIN, S. **Resource ReSerVation Protocol (RSVP).** V.1 Functional Specification, 1997. Disponível em: <https://tools.ietf.org/html/rfc2205>. Acesso em: 20 dez. 2016.

BRITO, I.; et al. **OpenWiMesh: A Framework for Software Defined Wireless Mesh Networks**. In: 2014 Brazilian Symposium on Computer Networks and Distributed Systems, Anais. IEEE, Maio 2014. Disponível em: <http://ieeexplore.ieee.org/document/6927136/>. Acesso em: 20 dez. 2016.

CARDOSO, R. F. F. **Uma abordagem SDN para o controlo e admissão de tráfego**. 2015. Disponível em: <https://run.unl.pt/bitstream/10362/16557/1/Cardoso_2015.pdf>. Acesso em: 21 dez. 2016.

CARLOS, C. C. **SDN (Software Defined Networks) | Thinking about Data Center SolutionsThinking about Data Center Solutions**. Disponível em: <http://blogs.salleurl.edu/data-center-solutions/2016/03/sdn-software-defined-networks-c2016/>. Acesso em: 21 dez. 2016.

CARROLL, B. W. **Konigsberg pridges problem**. Disponível em: <http://physics.weber.edu/carroll/honors/konigsberg.htm>. Acesso em: 21 dez. 2016.

CASADO, M.; et al. **Rethinking Enterprise Network Control**. IEEE/ACM Transactions on Networking, v. 17, n. 4, p. 1270–1283, ago. 2009. Disponível em: <http://ieeexplore.ieee.org/document/5169973/>. Acesso em: 20 dez. 2016.

CHOWDHURY, N. M. M. K.; BOUTABA, R. **Network virtualization: state of the art and research challenges.** IEEE Communications Magazine, v. 47, n. 7, p. 20–26, jul. 2009. Disponível em: <http://ieeexplore.ieee.org/document/5183468/>. Acesso em: 20 dez. 2016.

COHEN, R.; LEWIN-EYTAN, L.; NAOR, J. S.; RAZ, D. **On the effect of forwarding table size on SDN network utilization.** In: IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, Anais IEEE, abr. 2014. Disponível em: <http://ieeexplore.ieee.org/document/6848111/>. Acesso em: 21 dez. 2016.

CORRADI, A.; FANELLI, M.; FOSCHINI, L. VM **consolidation: A real case based on OpenStack Cloud**. Future Generation Computer Systems, v. 32, p. 118–127, mar. 2014. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0167739X12001082>. Acesso em: 21 dez. 2016.

DUQUE, D. H.; et al. **Redes definidas por software - Monografias.com**. Disponível em: <http://br.monografias.com/trabalhos3/redes-definidas-software/redes-definidas-software2.shtml>. Acesso em: 21 dez. 2016.

EARLEY, S.; HARMON, R.; LEE, M. R.; MITHAS, S. **From BYOD to BYOA, Phishing, and Botnets**. IT Professional, v. 16, n. 5, p. 16–18, set. 2014. Disponível em: <http://ieeexplore.ieee.org/document/6908910/>. Acesso em: 20 dez. 2016.

ERICKSON, D. **The beacon openflow controller**. Proceedings of the second ACM SIGCOMM workshop, p. 13–18, 2013. Disponível em: <http://dl.acm.org/citation.cfm?doid=2491185.2491189>. Acesso em: 21 dez. 2016.

FERNANDES, E. L.; ROTHENBERG, C. E. **OpenFlow 1.3 Software Switch.** Anais do 32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos – SBRC, p. 1021–1028, 2014.

FLOODLIGHT. **Floodlight OpenFlow Controller -Project Floodlight**, 2014. . Disponível em: <http://www.projectfloodlight.org/floodlight/>. Acesso em: 21 dez. 2016.

FOUNDATION, O. N. **Software-Defined Networking (SDN) Definition**. Disponível em: <https://www.opennetworking.org/sdn-resources/sdn-definition>. Acesso em: 20 dez. 2016.

FULLER, V.; LI, T.; YU, J.; VARADHAN, K. **Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy**. Disponível em: < https://tools.ietf.org/html/rfc1519 > Acesso em: 20 dez. 2016.

GAGNON, M. **Algoritmos e teoria dos grafos**. Disponível em: <http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/Grafos/Busca/busca.html>. Acesso em: 21 dez. 2016.

GÉANT. **What sets it apart? | GÉANT**. Disponível em: <http://www.geant.org/Networks/Pan-European_network/Pages/What-sets-it-apart.aspx>. Acesso em: 21 dez. 2016.

GORLATCH, S.; HUMERNBRUM, T.; GLINKA, F. **Improving QoS in real-time internet applications: from best-effort to Software-Defined Networks.** In: 2014 International Conference on Computing, Networking and Communications (ICNC), **Anais**...IEEE, fev. 2014.

Disponível em: <http://ieeexplore.ieee.org/document/6785329/>. Acesso em: 20 dez. 2016.

GUDE, N.; et al. **NOX: towards an operating system for networks**. SIGCOMM Computer Communication Review, v. 38, n. 3, p. 105–110, 1 jul. 2008. Disponível em: <http://portal.acm.org/citation.cfm?doid=1384609.1384625>. Acesso em: 21 dez. 2016.

GUEDES, D.; et al. **Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento das pesquisas em Redes de Computadores.** 2012 Disponível em: < http://homepages.dcc.ufmg.br/~mmvieira/cc/papers/minicurso-sdn.pdf>. Acesso em: 21 dez. 2016.

INTERNET2. **About Us | Internet2 - The Internet2 community: enabling the future**. Disponível em: <http://www.internet2.edu/about-us/>. Acesso em: 21 dez. 2016.

JIANG, J.-R.; et al. **Extending Dijkstra's shortest path algorithm for software defined networking**. In: The 16th Asia-Pacific Network Operations and Management Symposium, Anais IEEE, set. 2014. Disponível em: <http://ieeexplore.ieee.org/document/6996609/>. Acesso em: 21 dez. 2016.

KASIGWA, J.; BARYAMUREEBA, V.; WILLIAMS, D. **Dynamic Admission Control for Quality of Service in IP Networks**. In: Advances in Computer, Information, and Systems Sciences, and Engineering. Dordrecht: Springer Netherlands, 2007. p. 253–258.

KOBAYASHI, M.; et al. **Maturing of OpenFlow and Software-defined Networking through deployments**. Computer Networks, v. 61, p. 151–175, mar. 2014. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S138912861300371X>. Acesso em: 20 dez. 2016.

KREUTZ, D.; et al. **Software-Defined Networking: A Comprehensive Survey.** Proceedings of the IEEE, v. 103, n. 1, p. 14–76, jan. 2015. Disponível em: <http://ieeexplore.ieee.org/document/6994333/>. Acesso em: 21 dez. 2016.

KUROSE, James F.; ROSS, Keith W.; ZUCCHI, Wagner Luiz. **Redes de Computadores ea Internet: uma abordagem top-down**. Pearson, 2013.

LANTZ, Bob; HELLER, Brandon; MCKEOWN, Nick. **A network in a laptop: rapid prototyping for software-defined networks**. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM, 2010. p. 19.

MARTIÑA, Quiroz; JOSE, David. **Research on path establishment methods performance in SDN-based networks**. 2015. Dissertação de Mestrado. Universitat Politècnica de Catalunya.

MCKEOWN, Nick et al. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v. 38, n. 2, p. 69-74, 2008.

MELO, K. R. de. **Aplicações De Busca Em Grafos**. 2007. Disponível em:

<http://www.ceavi.udesc.br/arquivos/id_submenu/387/david_krenkel_rodrigues_de_melo.pdf>. Acesso em: 21 dez. 2016.

MIDDLETON, S. E.; MODAFFERI, S. **Scalable classification of QoS for real-time interactive applications from IP traffic measurements.** Computer Networks, v. 107, n. P1, p. 121–132, out. 2016. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1389128616301165>. Acesso em: 20 dez. 2016.

MILLINGTON, A. J.; et al. **The large hadron collider : the greatest adventure in town and ten reasons why it matters, as illustrated by the atlas experiment.** World Scientific Publishing Company, 2016.

NAVARRO, R. S.; et al. **Arquitetura de Transporte Inteligente baseada em SDN**. 2015. Disponível em: < https://www.researchgate.net/profile/Rodolfo_Meneguette/publication/280979990_Arquitetura_de_Transporte_Inteligente_baseada_em_SDN/links/56af4d9a08aeaa696f2fe2cf.pdf >. Acesso em: 20 dez. 2016

NOBRE, Tito Sérgio Martins Pereira. **SDN em rede de tranporte ópticas**. 2014. Tese de Doutorado. Disponível em: < http://repositorio.ul.pt/bitstream/10451/15936/1/ulfc112524_tm_Tito_Nobre.pdf >. Acesso em: 20 dez. 2016.

ONOS. **ONOS Tutorial | SDN Hub**. Disponível em: <http://sdnhub.org/tutorials/onos/>. Acesso em: 21 dez. 2016.

OPENDAYLIGHT. **Platform Overview | OpenDaylight**. Disponível em: <https://www.opendaylight.org/platform-overview>. Acesso em: 21 dez. 2016.

PANTUZA, Gustavo et al. **Análise e gerenciamento de rede através de grafos em redes definidas por software**. In: V WPEIF-Workshop de Pesquisa Experimental da Internet do Futuro. 2014. p. 17-20. Disponível em: < https://www.researchgate.net/profile/Dorgival_Guedes/publication/284719490_Analise_e_Gerenciamento_de_Rede_atraves_de_Grafos_em_Redes_Definidas_por_Software/links/5657099c08aeafc2aac0b448.pdf >. Acesso em: 21 dez. 2016.

RAGHAVENDRA, R.; LOBO, J.; LEE, K.-W. **Dynamic graph query primitives for SDN-based cloudnetwork management. In: Proceedings of the first workshop on Hot topics in software defined networks** - HotSDN '12, New York, New York, USA. Anais... New York, New York, USA: ACM Press, 2012. Disponível em: <http://dl.acm.org/citation.cfm?doid=2342441.2342461>. Acesso em: 21 dez. 2016.

REDE IPÊ. **Rede Ipê | RNP**. Disponível em: <https://www.rnp.br/servicos/conectividade/rede-ipe>. Acesso em: 21 dez. 2016.

REZENDE, P. H. A. **Extensões na arquitetura SDN para o provisionamento de QoS através do monitoramento e uso de múltiplos caminhos**. 2016. Disponível em: < https://repositorio.ufu.br/bitstream/123456789/17550/1/ExtensoesArquiteturaSDN.pdf > . Acesso em: 21 dez. 2016.

ROCHA, L. A.; VERDI, F. L. **MILPFlow: A toolset for integration of computational modelling and deployment of data paths for SDN.** In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), **Anais**...IEEE, maio 2015. Disponível em: <http://ieeexplore.ieee.org/document/7140367/>. Acesso em: 21 dez. 2016.

RYU. **RYU SDN Framework**. 2014. Disponível em: <http://osrg.github.io/ryu-book/en/Ryubook.pdf>. Acesso em: 21 dez. 2016.

SANDRI, M.; et al. **On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks.** In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, **Anais**...IEEE, mar. 2015. Disponível em: <http://ieeexplore.ieee.org/document/7097944/>. Acesso em: 21 dez. 2016.

SANDRI, M.; SILVA, A.; VERDI, F. L. **MultiFlow: Uma Solução para Distribuição de Subfluxos MPTCP em Redes OpenFlow**. 2015. Disponível em: < https://repositorio.ufscar.br/bitstream/handle/ufscar/637/SANDRI_Marcus_2015.pdf?sequence=1 &isAllowed=y > Acesso em: 21 dez. 2016.

SPALLA, E. S. **Universidade Federal do Espírito Santo Estratégias para Resiliência em SDN: Uma Abordagem Centrada em Multi-Controladores Ativamente Replicados**. 2015.

TAIRAKU, T.; et al. S**DN path control experiment based on social information by network virtualization node on JGN-X**. In: 2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR), Anais. IEEE, jun. 2016. Disponível em: <http://ieeexplore.ieee.org/document/7525651/>. Acesso em: 21 dez. 2016.

TOMOVIC, S.; PRASAD, N.; RADUSINOVIC, I. **SDN control framework for QoS provisioning**. In: Telecommunications Forum Telfor (TELFOR), 2014 22nd, Anais IEEE, nov. 2014. Disponível em: <http://ieeexplore.ieee.org/document/7034369/>. Acesso em: 20 dez. 2016.

TOMOVIC, S.; RADUSINOVIC, I.; PRASAD, N. **Performance comparison of QoS routing algorithms applicable to large-scale SDN networks. In: Proceedings** - EUROCON 2015, Anais. IEEE, set. 2015. Disponível em: <http://ieeexplore.ieee.org/document/7313698/>. Acesso em: 20 dez. 2016.

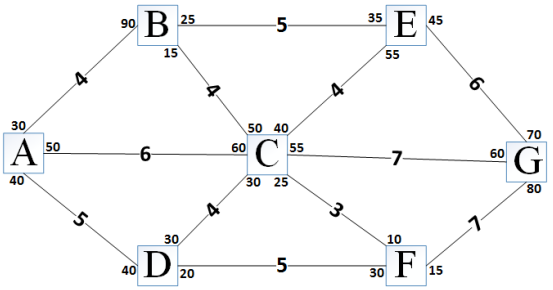
YAN, J.; et al. **An SDN-based multipath QoS solution**. China Communications, v. 12, n. 5, p. 123–133, maio 2015. Disponível em: <http://ieeexplore.ieee.org/document/7112035/>. Acesso em: 21 dez. 2016.

YILMAZ, S.; TEKALP, A. M.; UNLUTURK, B. D. **Video streaming over software defined networks with server load balancing.** In: 2015 International Conference on Computing, Networking and Communications (ICNC), Anais. IEEE, fev. 2015. Disponível em:

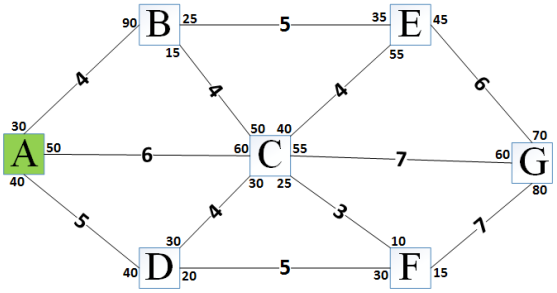
<http://ieeexplore.ieee.org/document/7069435/>. Acesso em: 20 dez. 2016.

8 ANEXOS

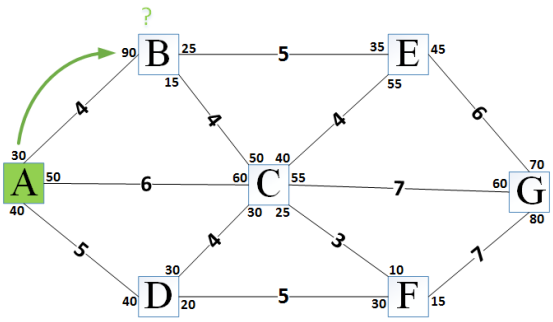
8.1 Fluxo de execução do algoritmo



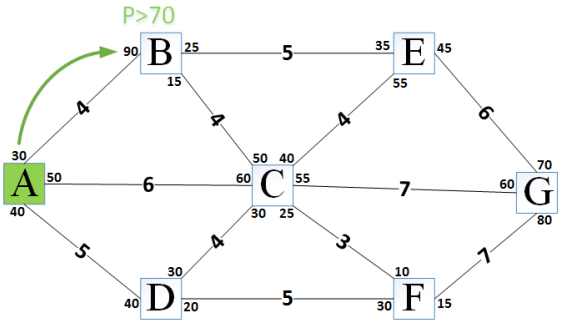
Q = A



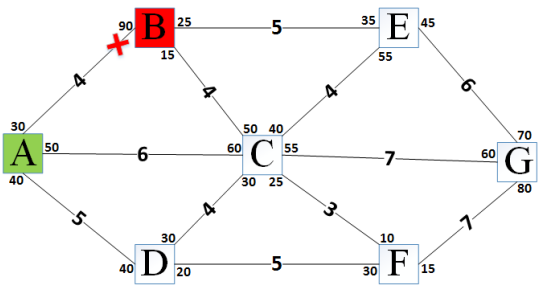
Q =



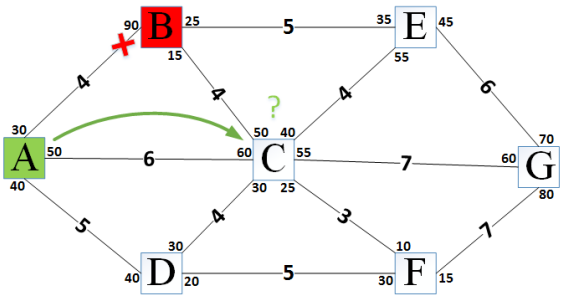
Q =



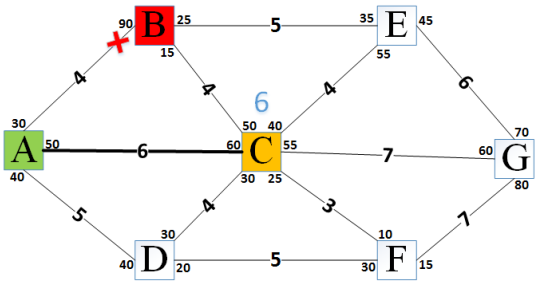
Q =



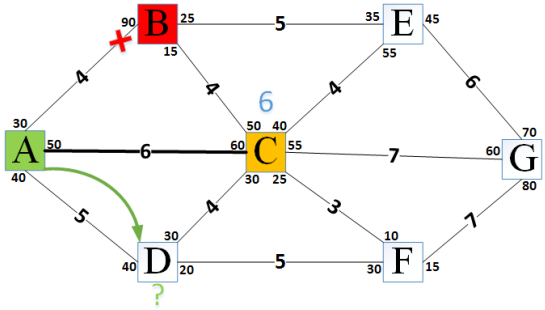
Q =



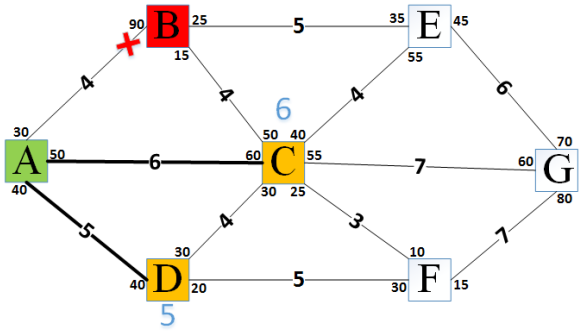
Q =



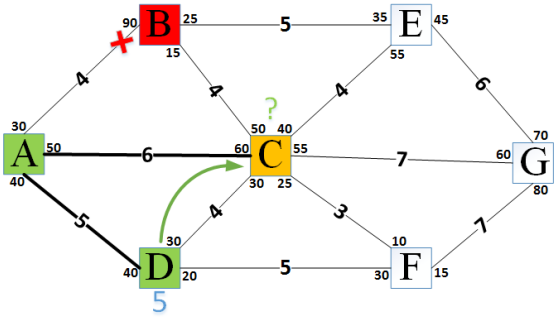
Q = C



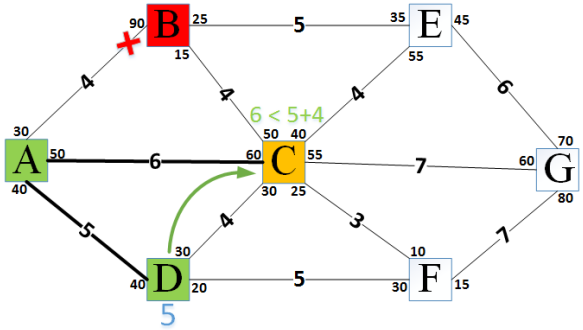
Q = C



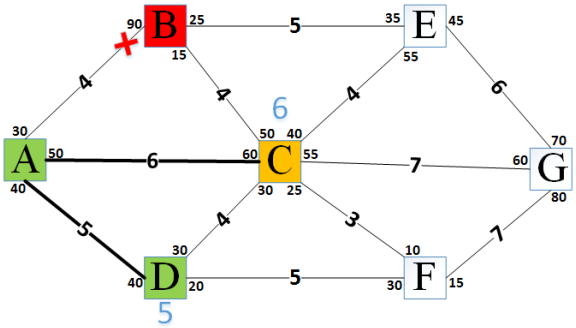
Q = D C



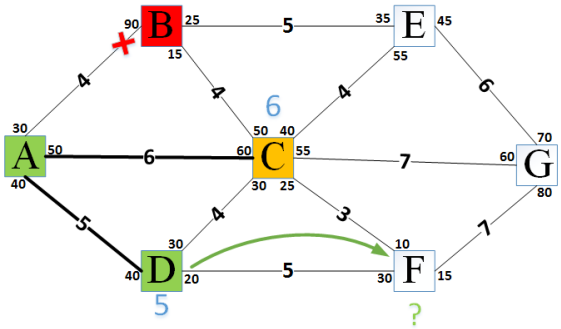
Q = C



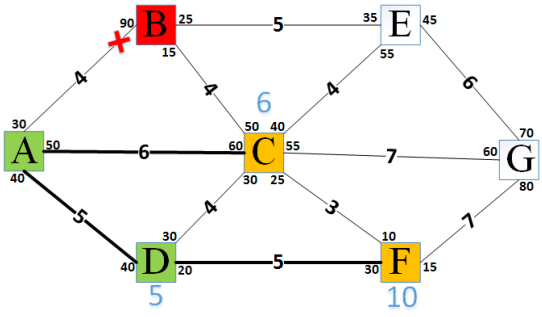
Q = C



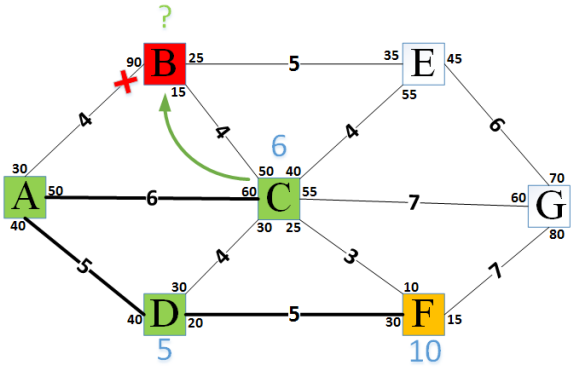
Q = C



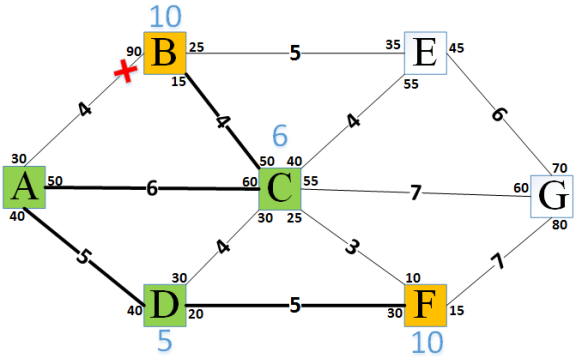
Q = C



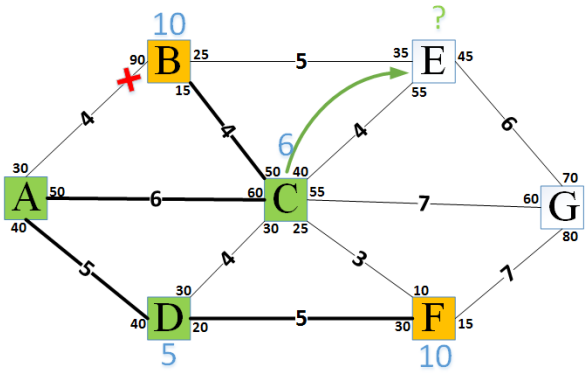
Q = C F



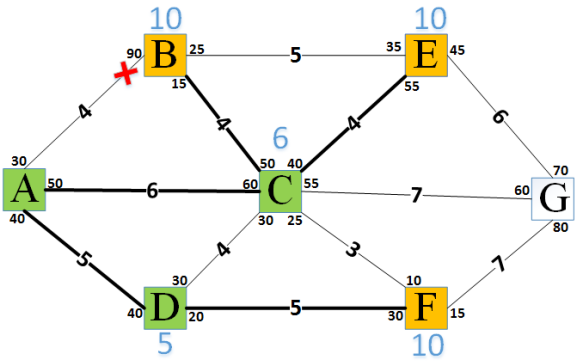
Q = F



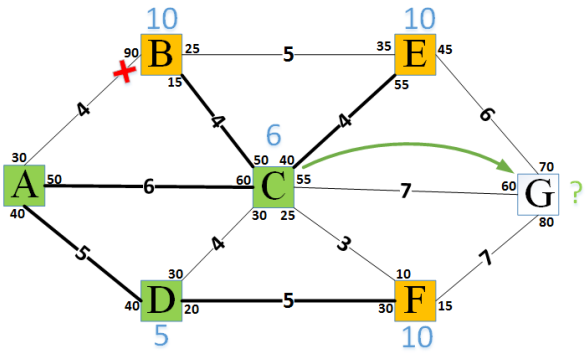
Q = F B



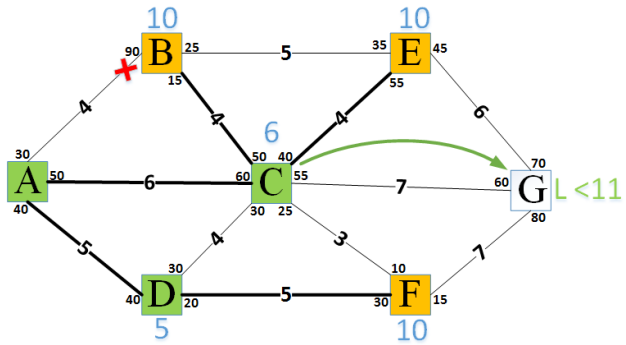
Q = F B



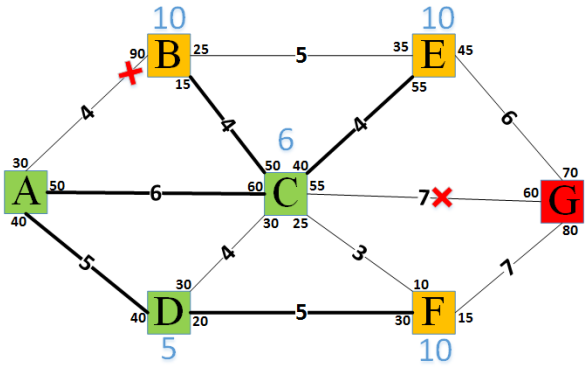
Q = F B E



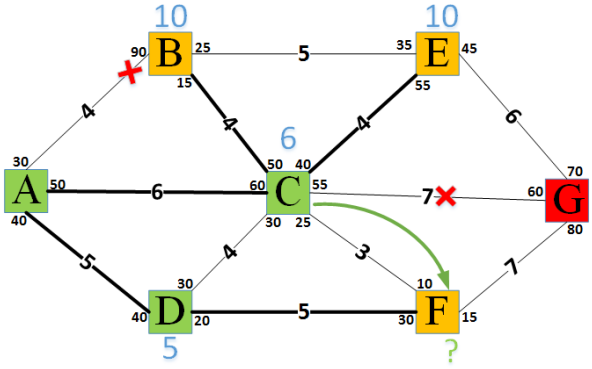
Q = F B E



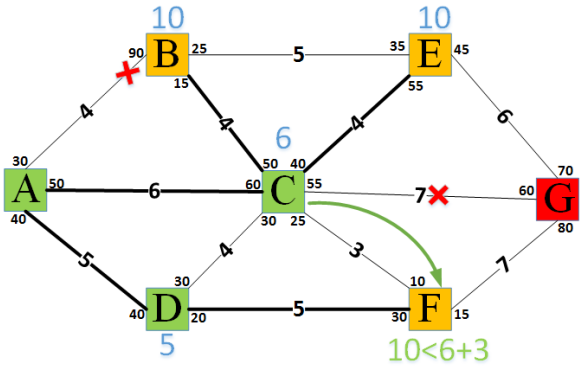
Q = F B E



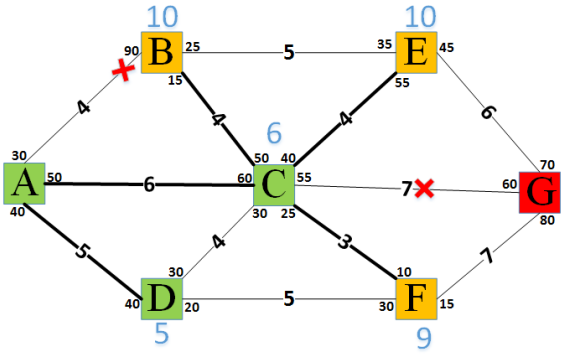
Q = F B E



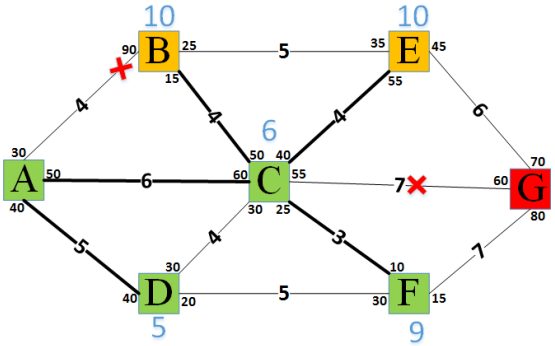
Q = F B E



Q = F B E

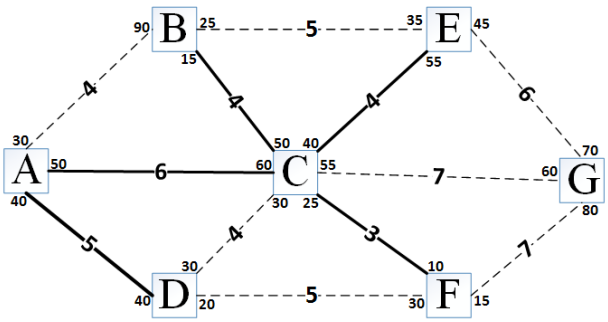


Q = F B E

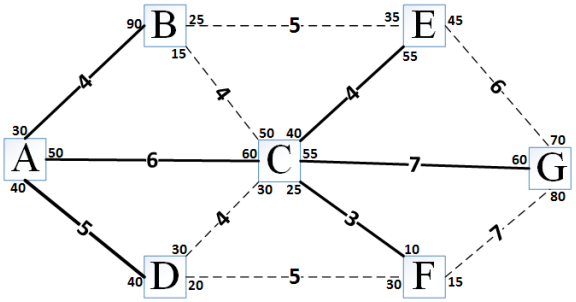


Q = B E

Árvore Geradora



Árvore Geradora Dijkstra



8.2 Exemplo de implementação do algoritmo em Java

```
public CaminhoVL getCaminho(GrafoVL g, Vertice origem, Vertice destino, double limite, int bloqueio) {
        grafo = g;
        List<Vertice> naoVisitados = new ArrayList<Vertice>();
        Vertice atual = new Vertice();
        Vertice vizinho = new Vertice();
        List<Vertice> menorCaminho = new ArrayList<Vertice>();
        Vertice verticeCaminho = new Vertice();
        boolean temCaminho = false;
        CaminhoVL caminho = new CaminhoVL();
        menorCaminho.add(origem);
        for (Vertice vertice : grafo.getVertices()) {
            if (vertice.getId_vertice() == origem.getId_vertice()) {
                vertice.setDistanciaOrigem(0);
            } else {
                vertice.setDistanciaOrigem(9999999);
            }
        }
        naoVisitados.add(origem);
        while (!naoVisitados.isEmpty()) {
            atual = naoVisitados.get(0);
            if (atual.getId_vertice() == destino.getId_vertice()) {
                break;
            }
            for (Link link : atual.getArestas()) {
                if (link.getProbBloqueio() < bloqueio) {
                    vizinho = link.getDestino();
                    if (!vizinho.isVerificado()) {
                        if ((atual.getDistanciaOrigem() + link.getAtraso()) < limite) {
```

```
                                        if (!vizinho.isNaListaNVisitados()) {
                                            naoVisitados.add(vizinho);
                                            vizinho.setNaListaNVisitados(true);
                                        }
                                        if (vizinho.getDistanciaOrigem() > (atual.getDistanciaOrigem() + link.getAtraso())) {

            vizinho.setDistanciaOrigem(atual.getDistanciaOrigem()+ link.getAtraso());
                                            vizinho.setPai(atual);
                                            if (vizinho.getId_vertice() == destino.getId_vertice()) {
                                                menorCaminho.clear();
                                                verticeCaminho = vizinho;

            menorCaminho.add(vizinho);
                                                while (verticeCaminho.getPai() != null) {

            menorCaminho.add(verticeCaminho.getPai());
                                                    verticeCaminho = verticeCaminho

            .getPai();
                                                }
                                                temCaminho = true;
                                            }
                                        }
                                    }
                                }
                            }
                            atual.setVerificado(true);
```

```
                naoVisitados.remove(0);
                Collections.sort(naoVisitados);
            }
            caminho.setMenorCaminho(menorCaminho);
            caminho.setCaminho(temCaminho);
            return caminho;

    }
```