UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DIOGO MACHADO GONÇALVES

ANÁLISE DE FILAS DE PRIORIDADES PARA O ALGORITMO DE DIJKSTRA EM REDES DE MALHAS SEM FIO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2015

DIOGO MACHADO GONÇALVES

ANÁLISE DE FILAS DE PRIORIDADES PARA O ALGORITMO DE DIJKSTRA EM REDES DE MALHAS SEM FIO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná.

Orientadora: Dra. Sheila Morais de Almeida

Co-orientador: MSc. Saulo Jorge Beltrão de Queiroz

PONTA GROSSA

2015

TERMO DE APROVAÇÃO

ANÁLISE DE FILAS DE PRIORIDADE PARA O ALGORITMO DE DIJKSTRA EM REDES DE MALHA SEM FIO

por

DIOGO MACHADO GONÇALVES

Este(a) Trabalho de Conclusão de Curso (TCC) foi apresentado(a) em 10 de Novembro de 2015 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____	_____
Dra. Sheila Morais de Almeida Profa. Orientadora	Dra. Tânia Lúcia Monteiro Membro titular
	_____
	Dr. Gleifer Vaz Alves Membro titular
_____	_____
Dr. Ionildo José Sanches Responsável pelos Trabalhos de Conclusão de Curso	Dr. Erikson Freitas De Morais Coordenador do Curso UTFPR – Campus Ponta Grossa

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedico este trabalho aos meus pais, pelo apoio e motivação.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ser meu guia nos momentos difíceis e companhia durante a solidão. A ele toda a honra e toda a glória.

Agradeço aos meus pais, Luisa e Osmar, pelo apoio e confiança que depositaram em mim, não somente durante estes 4 anos mas desde que tenho consciência da vida.

Sou eternamente grato aos meus tios Ulisses e Isabel por todo o apoio e cuidado que me ofereceram, não medindo esforços para que a minha adaptação a esta etapa da vida que se encerra agora fosse feita da melhor forma possível. Também considero vocês como pais.

Agradeço àquele com quem tive o privilégio de conviver por quase 3 anos. Ao meu orientador Saulo Queiroz pelos ensinamentos e conselhos dados durante este período. Com certeza um exemplo pessoal e profissional que levarei por toda a vida.

Àquela com quem sempre sonhei um dia trabalhar, mas o convívio com o Saulo e a satisfação em pesquisar com ele não me deixavam mudar de caminho. Quis Deus permitir que eu trabalhasse com a Sheila sem precisar abandonar ninguém. Não poderia haver um desfecho melhor, tive os melhores orientadores que um aluno poderia ter.

A todos os professores pelos ensinamentos passados, me fazendo amar ainda mais minha profissão.

Agradeço especialmente aos meu amigos Breno, Gisele, Letícia, Jonas, Lucas, Bruno e Jônatas pelos momentos que passamos durante estes 4 anos, sem a presença de vocês esta caminhada teria sido muito mais difícil.

Um agradecimento especial também ao meu amigo Anderson pelo apoio e conselhos dados, mostrando que mesmo à distância continuamos melhores amigos.

Aos meus primos Juarez e Adílson pelo apoio e companhia, e a toda minha família e amigos que de alguma forma contribuíram para a minha formação.

É fazendo qualquer coisa que você se
torna qualquer um.
(GAILLARD, Remi)

RESUMO

GONCALVES, Diogo Machado. Análise de filas de prioridades para o Algoritmo de Dijkstra em redes de malhas sem fio: 2015. 110 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

Garantir a escalabilidade de um protocolo de roteamento de redes de malha sem fio contribui para a manutenção da qualidade da conexão oferecida pela rede com a adição de novos clientes. Otimizar o Algoritmo de Dijkstra utilizado pelo protocolo OLSRD para calcular as rotas que os pacotes de dados da rede devem seguir contribui significativamente para o aumento do desempenho da rede. Este trabalho apresenta um estudo empírico acerca do impacto de diferentes filas de prioridades na complexidade do Algoritmo de Dijkstra. Com base nas avaliações teóricas e práticas de desempenho da fila de prioridade do protocolo de redes de malha sem fio OLSRD, outras estruturas de dados foram selecionadas na literatura para uma avaliação de desempenho. Filas de prioridades em cenários característicos de redes de malha sem fio apresentaram um grande número de itens com prioridades repetidas inseridos nestas estruturas de dados, prejudicando a árvore AVL, estrutura presente no protocolo e contribuindo para a escolha do heap binário e Van Emde Boas como as melhores filas de prioridade para grandes redes neste contexto.

Palavras-chave: Algoritmo de Dijkstra. Filas de prioridade. Redes de malha sem fio. OLSRD.

ABSTRACT

GONCALVES, Diogo Machado. Complexity analysis of Dijkstra’s priority queues on wireless mesh networks: 2015. 110 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Federal Technology University - Parana. Ponta Grossa, 2015.

A scalable wireless mesh network protocol helps to keep a good quality of service when new clients are added. An improvement on Dijkstra's Algorithm used by the OLSRD protocol to find the shortest paths for the data packages through the network, increases the performance of the network. This work shows an empirical analysis about the Dijkstra's performance with different priority queues. On the basis of theoretical and practical performance evaluations in the OLSRD's priority queue, other data structures were selected to be compared with it. In a mesh network scenario, Dijkstra's Algorithm inserts a big amount of items with the same key in the priority queue, which reduces AVL tree performance. Based on this fact, this work shows that the binary heap and Van Emde Boas tree are a better choice compared to the AVL tree, current priority queue used in OLSRD.

Keywords: Dijkstra's Algorithm. Priority queue. Wireless Mesh Network. OLSRD.

LISTA DE ILUSTRAÇÕES

LISTA DE GRÁFICOS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AP	Access Point
AVL	Árvore Adelson-Velsky e Landis
BATMAN	Better Approach To Mobile Adhoc Networks
BFS	Breadth First Search
BH	Binary Heap
BST	Binary Search Tree
DSSSP	Dynamic Single Source Shortest Paths
ETX	Expected Transmission Count
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
LSP	Link State Packet
MID	Multiple Interface Declaration
MPR	Multipoint Relay
OLSR	Optimized Link State Routing
OLSRD	Optimized Link State Routing Daemon
SSSP	Single Source Shortest Paths
TTL	Time To Live
VEB	Van Emde Boas
WLAN	Wireless Local Area Network
WMN	Wireless Mesh Network

LISTA DE SÍMBOLOS

O	Limite assintótico superior
Ω	Limite assintótico inferior
Θ	Limite assintótico restrito
o	Limite assintótico superior de forma firme
ω	Limite assintótico inferior de forma firme
u	Número máximo de valores distintos suportado pela Van Emde Boas.
Q	Conjunto de itens na fila de prioridade.
V	Conjunto de vértices
A	Conjunto de arestas
n	Número de itens
$\|V\|$	Cardinalidade (número de elementos) do conjunto V
$\|A\|$	Cardinalidade (número de elementos) do conjunto A

SUMÁRIO

1 INTRODUÇÃO

A popularização de dispositivos compatíveis com o padrão IEEE 802.11 (IEEE, 2012) como computadores portáteis e *smartphones* têm contribuído para o crescimento de *hotspots*, áreas conhecidas por disponibilizar serviços Wi-Fi (*Wireless-Fidelity*), a fim de oferecer ao usuário uma conexão sem fio à Internet.

A expansão de redes locais sem fio (*Wireless Local Area Network* - WLAN) requer investimento em infraestrutura cabeada, além da aquisição de pontos de acesso (*Access Point* - AP), responsáveis por permitir que o usuário se conecte à rede a partir de tecnologias sem fio.

Visando facilitar o processo de expansão do raio de cobertura das WLANs, as redes de malha sem fio (*Wireless Mesh Network* - WMN), empregam APs de baixo custo para a implementação de roteadores que encaminham dados entre si através de múltiplos saltos pela tecnologia sem fio, como visto em Frangoudis, Polyzos e Kemerlis (2011) e Siris, Tragos e Petroulakis (2012).

Em WMNs, o encaminhamento de dados entre os roteadores é gerenciado por um protocolo de roteamento. Nesse contexto, a *Internet Engineering Task Force* (IETF) padronizou o protocolo de roteamento de estado de enlace global chamado *Optimized Link State Routing* (OLSR) através da RFC 3626 (CLAUSEN; JACQUET, 2003).

O protocolo proativo OLSR representa dinamicamente a topologia de uma WMN através de trocas periódicas de mensagens de controle contendo informações topológicas dos roteadores. Após processar cada uma dessas mensagens, o roteador executa um algoritmo para calcular os caminhos de custo mínimo para os demais roteadores, a fim de atualizar a tabela de roteamento, responsável por indicar as rotas de encaminhamento dos pacotes de dados da rede. Como um protocolo proativo, o OLSR mantém as rotas atualizadas para serem utilizadas assim que requisitadas. Mais especificamente, a implementação OLSRD (*Optimized Link State Routing Daemon*) do protocolo utiliza o Algoritmo de Dijkstra (1959) para a função do cálculo de caminhos mínimos.

A capacidade da rede em se autoconfigurar mediante a inclusão ou remoção de um ponto de acesso facilita o rápido crescimento da rede sem um alto custo de projeto. Por outro lado, tal característica, aliada aos escassos recursos computacionais oferecidos pelos roteadores, pode comprometer severamente o desempenho do algoritmo de caminho mínimo devido ao aumento do volume de dados a serem processados. Medições em uma WMN real com 450 APs mostraram que uma implementação do Algoritmo de Dijkstra consumia vários segundos para atualizar a tabela de roteamento (OPEN-MESH, 2014).

Em face deste problema, alternativas têm sido investigadas a fim de diminuir a complexidade média para solução do problema. No estudo proposto por Queiroz (2009), foi verificado que o algoritmo baseado no Algoritmo de Dijkstra proposto por Ramalingam e Reps (1996) apresentou um melhor desempenho comparado com o algoritmo original para o caso médio de uma WMN, contrariando o resultado da comparação de pior caso entre os algoritmos. O referido estudo trouxe grandes contribuições para o desempenho da rede, porém não abrangeu a escolha das filas de prioridades utilizadas pelo Algoritmo de Dijkstra e pelo Algoritmo de Ramalingan e Reps.

Considerando que as filas de prioridades cumprem papel determinante na complexidade desses algoritmos de caminho mínimo, este trabalho propõe um estudo empírico a fim de avaliar o impacto de diferentes filas de prioridades na complexidade do Algoritmo de Dijkstra, que é base para o cálculo de rotas no contexto de redes de malha sem fio.

1.1 OBJETIVOS

A seguir serão descritos os objetivos gerais e específicos deste trabalho.

1.1.1 Objetivo Geral

Este trabalho tem por objetivo apontar a fila de prioridade, aplicada no Algoritmo de Dijkstra, mais eficiente no contexto de redes de malha sem fio em um protocolo proativo.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Definir as funções assintoticamente relevantes da árvore AVL, a fila de prioridade atualmente implementada no Algoritmo de Dijkstra do protocolo proativo OLSRD;
- Pesquisar as filas de prioridade disponíveis na literatura com suporte à métrica utilizada no protocolo e que apresentam um desempenho teórico competitivo comparado com a árvore AVL;
- Implementar um heap binário e analisar o desempenho de suas operações

de filas de prioridades considerando a dinâmica de uma rede de malha sem fio;

- Adaptar uma árvore Van Emde Boas como fila de prioriodade e analisar o desempenho de suas operações considerando a dinâmica de uma rede de malha sem fio;
- Otimizar o desempenho das filas de prioridade estudadas para o contexto do protocolo OLSRD;
- Desenvolver um ambiente de teste que permita a avaliação das estruturas de dados em grafos que apresentam características de uma rede de malha sem fio;
- Analisar o desempenho do Algoritmo de Dijkstra utilizando as filas de prioridade abordadas em um ambiente com as características de uma rede de malha sem fio;
- Implementar no protocolo OLSRD a fila de prioridade com o melhor desempenho obtido, considerando os resultados relativos ao consumo de tempo e espaço.

1.2 JUSTIFICATIVA

No contexto de uma WMN, devido à periódica troca de mensagens a fim de manter a topologia da rede atualizada, o recálculo das rotas ótimas pode vir a ser recorrente. Este recálculo pode comprometer a eficiência da rede, principalmente considerando a capacidade limitada de processamento dos APs. Garantir a escalabilidade de uma rede de malha sem fio é essencial para prover um serviço de qualidade com a adição de novos clientes. Rotear rapidamente os pacotes da rede é um dos requisitos envolvidos na escalabilidade, requerendo algoritmos eficientes para tal tarefa.

Uma solução clássica para o cálculo de rotas mínimas em redes consiste no uso do Algoritmo de Dijkstra, cuja complexidade depende da estrutura de dados de fila de prioridades necessária ao seu funcionamento. Por sua vez, o desempenho do algoritmo sob uma determinada fila de prioridades depende das características do grafo de entrada que, nesse caso, são determinadas pelas características da topologia da rede sem fio.

Para diminuir o processamento investido pelos APs para manter a tabela de roteamento atualizada, estudos foram realizados para amenizar este problema. Um exemplo de otimização da manutenção da tabela de roteamento é apresentado por Queiroz (2009), que propõe o uso do modelo de computação incremental limitada

através da implementação do Algoritmo de Ramalingan e Reps para a manutenção desta tabela.

Apesar do Algoritmo de Dijkstra se caracterizar como a melhor solução para o problema do menor caminho de origem única, o contexto de uma rede *mesh* transforma o problema em uma variação conhecida como o problema do menor caminho de origem única dinâmico (Dinamic Single Souce Shortest Path - DSSSP), dadas as constantes mudanças no grafo da rede decorrentes das alterações nas conexões entre os roteadores, além das eventuais inserções e remoções de roteadores na rede.

O padrão do protocolo OLSR sugere que, a cada mudança topológica da rede, ou seja, alterações no grafo, o cálculo dos caminhos mínimos deve ser refeito para atualizar a tabela de roteamento. Os resultados da pesquisa porém, indicaram que a maioria das mudanças topológicas da rede não alteravam a tabela de roteamento, mesmo em variados cenários. O trabalho de Queiroz (2009) conclui que muitas vezes o recálculo das rotas é feito de forma desnecessária, já que os resultados são os mesmos previamente calculados.

O modelo de computação incremental propõe que o recálculo seja feito apenas quando a mudança topológica apresentar alguma alteração na tabela de roteamento, além de efetuar o recálculo apenas para o subgrafo afetado por tal mudança. Tal propriedade aplicada no contexto de uma WMN fez o Algoritmo de Ramalingan e Reps apresentar uma desempenho melhor que o Algoritmo de Dijkstra no caso médio.

O estudo trouxe grandes contribuições para o desempenho da rede, porém não abrangeu a escolha da fila de prioridades mais eficiente para o Algoritmo de Dijkstra no contexto de rede em malha sem fio. Identificar a estrutura de dados ideal para este cenário também é fundamental para melhorar a escalabilidade da rede.

Oberhauser e Simha (1995) por outro lado, apresentaram um estudo do comportamento do Algoritmo de Dijkstra a partir da utilização de diferentes filas de prioridades. Baseado nas estatísticas de desempenho do Algoritmo de Dijkstra em grafos com diferentes características, o artigo apresenta uma comparação de tempo de execução das filas de prioridade heap binário, heap binomial, *relaxed heap*, *run relaxed heap*, heap de Fibonacci e árvore splay. O trabalho de Oberhauser e Simha (1995) porém, apresenta uma análise de desempenho em contextos genéricos, não considerando a dinâmica apresentada por uma rede de malha sem fio.

Efetuar uma análise de desempenho do Algoritmo de Dijkstra considerando a utilização de variadas estruturas de dados no contexto de uma WMN possibilita apontar a fila de prioridade mais eficiente em tal cenário. A identificação das melhores estruturas de dados para este problema contribui para a diminuição do tempo de resposta do cálculo das rotas e, consequentemente, contribui para a eficiência do tempo de entrega de pacotes na rede.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado da seguinte forma.

O Capítulo 2 faz uma introdução às redes de malha sem fio e às filosofias de roteamento utilizadas pelos protocolos neste contexto. A partir desta contextualização, é descrito o funcionamento do protocolo OLSRD e apresentada uma introdução a um dos emuladores de rede de malha sem fio disponíveis para o protocolo descrito.

O Capítulo 3 apresenta uma visão geral acerca das estruturas de dados utilizadas pelo processo de cálculo de caminhos mínimos no protocolo OLSRD. Este capítulo descreve conceitos básicos necessários ao entendimento da organização interna do protocolo, além de detalhes de implementação do Algoritmo de Dijkstra e das estruturas de dados heap binário, árvore AVL e árvore Van Emde Boas, utilizadas como fila de prioridade pelo Algoritmo de Dijkstra neste estudo. O capítulo apresenta também uma breve descrição de conceitos básicos de análise de algoritmos utilizados para a comparação de desempenho das estruturas de dados.

O Capítulo 4 descreve os procedimentos realizados para a implementação e avaliação de desempenho das filas de prioridade bem como a escolha dos critérios para a construção dos cenários de teste dos algoritmos.

O Capítulo 5 apresenta o desempenho obtido pelas filas de prioridade nos cenários construídos para a avaliação dos algoritmos.

Por fim o Capítulo 6 descreve as conclusões deste trabalho assim como as recomendações de trabalhos futuros.

2 REDES DE MALHA SEM FIO

A possibilidade de oferecer ao usuário uma conexão sem fio à Internet tem contribuído para a popularização de pontos de acesso compatíveis com o padrão 802.11. Tradicionalmente, os APs constituem uma rede local sem fio (WLAN), atuando como interface entre o usuário e a infraestrutura de rede cabeada responsável pelo acesso à Internet.

Como ilustrado na arquitetura WLAN da Figura 1, cada AP é conectado individualmente a uma estrutura cabeada, criando-se a dependência de que cada ponto de acesso possua uma conexão direta à Internet para que os clientes conectados usufruam do serviço.

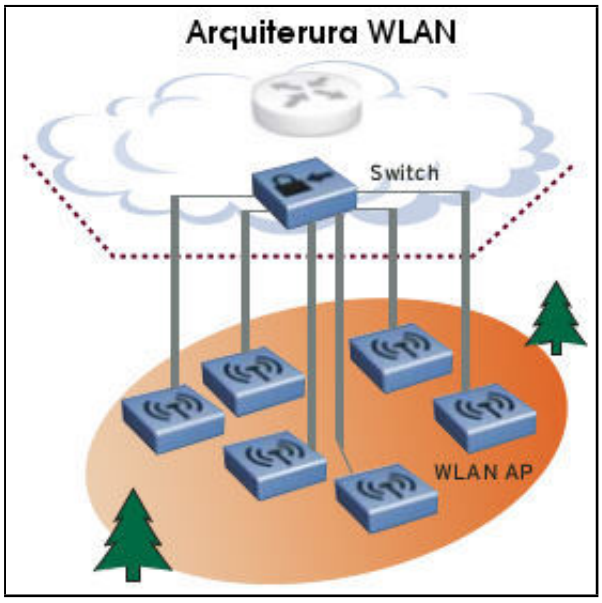

Figura 1 – Arquiterura WLAN
Fonte: Adaptado de Nortel (2006)

Esta dependência de infraestruturas individuais acarreta problemas de escalabilidade em grandes redes, visto que a adição de novos APs à rede aumenta a complexidade da infraestrutura, podendo levar ao planejamento de um novo projeto para atender o crescimento da mesma.

Como alternativa às limitações das WLANs, as redes em malha sem fio (*Wireless Mesh Network* - WMN) exploraram a capacidade de roteamento de APs (APs *mesh*) para interligá-los, podendo assim, encaminhar pacotes de dados entre os roteadores da rede. Caso um AP não possua uma infraestrutura que lhe permita acesso direto à Internet, ele encaminha os pacotes de seus clientes para os APs com acesso à Internet mais próximos.

A Figura 2 apresenta uma arquitetura de rede de malha sem fio, permitindo que todos os pontos de acesso da rede disponibilizem uma conexão à Internet para

seus clientes através do encaminhamento de pacotes para pelo menos um AP amparado com uma infraestrutura cabeada. Este processo é realizado pelo protocolo de roteamento embarcado em um *firmware* baseado em Linux instalado no aparelho, como por exemplo (FREIFUNK.NET, 2014) e (OPENWRT, 2014).

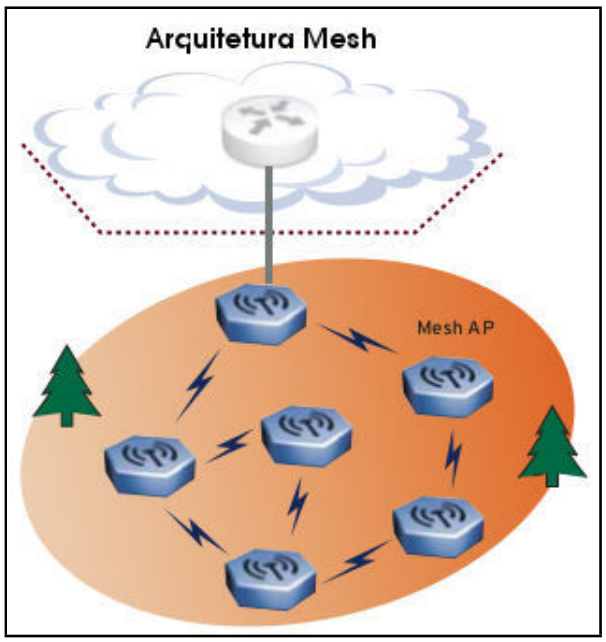

Figura 2 – Arquiterura de rede mesh
Fonte: Adaptado de Nortel (2006)

Pelo fato de não obrigar todos os APs da rede à estarem conectados diretamente à Internet, as WMN se tornam uma alternativa mais viável, visto que não necessitam de uma infraestrutura cabeada individual para garantir o serviço em todos os pontos de acesso.

Outra vantagem apresentada pelas WMN é sua capacidade de autoconfiguração, onde o protocolo de roteamento identifica a existência de novos APs e os insere na rede. Esta capacidade também é útil ao manter a consistência da rede com a saída inesperada de algum aparelho.

Devido a essas características, como baixo custo de implementação e fácil expansão, vários estudos foram feitos comprovando a eficiência das redes *Mesh* em expandir a cobertura de sinais Wi-Fi em variados locais, como centros urbanos (FRANGOUDIS; POLYZOS; KEMERLIS, 2011; SIRIS; TRAGOS; PETROULAKIS, 2012) e universidades (YU; XU; WU, 2010), além de auxiliar aplicações industriais, como visto em (LINDHORST; LUKAS; NETT, 2013). Alguns exemplos de projetos de redes *Mesh* comunitárias são vistos em (AWMN, 2014; SEATTLE WIRELESS, 2014; FREIFUNK, 2014).

Este capítulo tem como objetivo apresentar uma introdução às redes de malha sem fio, descrever as principais características dos protocolos utilizados neste contexto, e em especial o protocolo OLSR e um dos emuladores para a implementação

OLSRD. A Seção 2.1 apresenta as filosofias de roteamento dos protocolos de rede utilizados pelas redes de malha sem fio. A Seção 2.2 descreve as características do protocolo OLSR, como o processo para a descoberta da topologia da rede e as métricas utilizadas para a avaliação das conexões entre os roteadores. A Seção 2.3 descreve o funcionamento do emulador Olsr_switch, desenvolvido para a implementação OLSRD.

2.1 PROTOCOLOS DE ROTEAMENTO PARA REDES EM MALHA SEM FIO

A fim de prover o encaminhamento de pacotes entre os pontos de acesso da rede, os protocolos de roteamento são os responsáveis por gerenciar o cálculo do caminho entre a origem e o destino dos pacotes. Segundo Erciyes (2013, p. 100) “roteamento em uma rede de computadores é o processo de comunicação de mensagens de nós de origem para nós de destino por caminhos selecionados de acordo com os menores custos possíveis”.

Dadas as características apresentadas pelas WMN, diferentes critérios são utilizados na classificação de protocolos de roteamento neste contexto. Tais critérios podem considerar quais informações serão trocadas, quando e como estas informações de roteamento serão transferidas, e quando o cálculo de rotas será efetuado (ERCIYES, 2013). A partir de tais critérios, as redes de malha sem fio têm aplicado duas filosofias de roteamento (reativa e proativa) adotadas pela IETF.

2.1.1 Filosofias de Roteamento de Internet

Apesar dos protocolos tradicionalmente utilizados para roteamento de redes (Protocolo de Estado de Enlace e Protocolo de Vetor Distância) não apresentarem características necessárias para serem utilizados em redes sem fio (ERCIYES, 2013), os principais protocolos de roteamento desenvolvidos para este contexto apresentam características herdadas destes protocolos.

2.1.1.1 Protocolo de estado de enlace

O protocolo de estado de enlace propõe que os nós (roteadores) da rede troquem mensagens de controle entre si a fim de difundirem informações topológicas da rede. A partir de tais mensagens, cada AP consegue construir uma visão geral da rede, podendo executar um algoritmo de cálculo de caminhos mínimos para determi-

nar a melhor rota para cada destino da rede.

As mensagens topológicas da rede são trocadas periodicamente. Cada AP divulga para a rede um Pacote de Estado de Enlace (*Link State Packet* - LSP) contendo basicamente, seu identificador, a lista de seus vizinhos com os respectivos custos dos enlaces e um campo especial chamado Tempo De Vida (*Time To Live* - TTL), que indica a validade do pacote (ERCIYES, 2013).

Assim que um roteador recebe um pacote LSP ele verifica se o pacote possui informações mais atualizadas que as que ele tem conhecimento. Caso isto ocorra, o AP divulga tais informações para a rede.

Segundo Erciyes (2013), o processo de construção da topologia da rede por parte de cada roteador pode ser especificado pela seguinte metodologia:

1. Enviar Pacotes de Estado de Enlace periodicamente para seus vizinhos;
2. Quando um pacote recebido trouxer informações atualizadas acerca da rede:
    a) Recalcular a rota para os demais nós da rede;
    b) Armazenar o destino e o próximo salto referentes a cada nó da rede na tabela de roteamento.
3. Se o Pacote recebido não apresentar informações atualizadas, descarte-o.

2.1.1.2 Protocolo de vetor distância

Diferentemente da metodologia aplicada pelo protocolo de estado de enlace, os roteadores baseados no protocolo de vetor distância não possuem o conhecimento de toda a topologia da rede. Para que um AP possua uma estimativa do melhor caminho para todos os nós da rede, cada roteador divulga os valores dos melhores caminhos para os nós conhecidos por ele, ou seja, o resultado do cálculo propriamente dito.

Cada AP mantém uma tabela com as informações referentes ao destino para o qual os pacotes devem ser encaminhados e a distância até o referido nó. A partir de constantes trocas de mensagens, cada roteador divulga tais informações e atualiza suas próprias através das mensagens recebidas.

2.1.2 Protocolo Reativo

Protocolos de roteamento reativos constroem as rotas dos destinos da rede apenas quando requisitadas. Sempre que um roteador necessita enviar um pacote de dados para um destino na rede ele inicia o processo de descoberta de uma rota para tal destino, verificando o caminho em sua tabela de roteamento (caso exista) ou divulgando na rede uma solicitação de rota.

O processo de solicitação consiste em enviar mensagens para os nós da rede contendo informações do endereço de destino a fim de que algum nó informe uma rota para o AP requisitado.

Quando o nó recebe uma mensagem de solicitação de rota, ele verifica se sua tabela de roteamento possui uma rota para o destino informado. Caso exista, o AP envia esta rota para o roteador requisitante e o processo é encerrado. Caso contrário, o pacote recebido pelo AP é encaminhado para seus vizinhos, continuando o processo de busca pela rota (ERCIYES, 2013).

Sempre que o processo de solicitação de rota é executado, a rota encontrada é mantida temporariamente na tabela de roteamento, a fim de evitar mensagens de solicitações consecutivas para um mesmo destino em um curto período de tempo.

2.1.3 Protocolo Proativo

Diferentemente dos protocolos reativos, os protocolos proativos de roteamento mantêm as rotas para todos os destinos da rede antes que sejam solicitadas. Sempre que um roteador detectar uma mudança topológica na rede que julgue relevante, o cálculo das rotas é refeito a fim de mantê-las atualizadas. O objetivo é permitir que as rotas sejam utilizadas imediatamente quando requisitadas.

Cada roteador divulga periodicamente informações topológicas dos nós da rede que ele tem conhecimento, e não o resultado do cálculo das rotas entre eles. A partir de tais mensagens, o AP permite que os demais nós da rede atualizem as rotas para seus destinos.

Considerando o dinamismo das WMN, a divulgação de mensagens topológicas pode levar a rede a consumir toda sua capacidade de banda (TONNESEN, 2004), visto que o processo de divulgação destas mensagens independe de solicitações dos demais APs.

2.2 PROTOCOLO OPTIMIZED LINK STATE ROUTING - OLSR

O *Optimized Link State Routing* (OLSR) é um protocolo de roteamento proativo pertencente à filosofia de estado de enlace. Em 2004 o OLSR ganhou uma implementação escrita em linguagem C por Tonnesen (2004) denominada OLSRD (*Optimized Link State Routing Daemon*) (OLSR.ORG, 2015), vindo a ser a principal implementação do protocolo.

Através de mensagens de controle trocadas periodicamente entre os roteadores o protocolo constrói um grafo refletindo a visão global da rede. De posse destes dados topológicos, o protocolo executa um algoritmo de cálculo de caminhos mínimos para determinar uma rota que ofereça o menor custo (de acordo com a métrica utilizada) para transportar os pacotes de dados do ponto de acesso em questão para os demais. O resultado deste cálculo atualiza a tabela de roteamento de cada *access point*. O OLSRD especificamente utiliza o Algoritmo de Dijkstra como o algoritmo de cálculo de caminhos mínimos.

A fim de diminuir o volume de mensagens topológicas transmitidas na rede, o OLSR utiliza um mecanismo chamado *Multipoint Relay* (MPR), em que cada nó seleciona alguns de seus vizinhos para formar um conjunto de roteadores MPR. No OLSR, somente nós selecionados como MPRs são responsáveis por repassar mensagens de controle. Roteadores não selecionados como MPR podem processar as mensagens topológicas recebidas, porém não possuem autorização de retransmiti-las na rede (CLAUSEN; JACQUET, 2003).

Um conjunto MPR deve ser construído de tal forma que permita a todo roteador alcançar qualquer um de seus vizinhos a dois saltos de distância através de um nó MPR (TONNESEN, 2004). A Figura 3 ilustra a divulgação de mensagens topológicas a partir do roteador central utilizando o método tradicional. Cada seta representa uma transmissão. A Figura 4 ilustra o processo de divulgação utilizando nós MPR, apresentados em destaque. O número de mensagens transmitidas na rede diminui significativamente utilizando a abordagem *Multipoint Relay* (TONNESEN, 2004).

O processo de descoberta topológica da rede é formado a partir da troca de mensagens entre os roteadores da rede. Neste sentido, o protocolo define três mensagens de controle responsáveis por atualizar a tabela de roteamento, sendo elas, as mensagens HELLO, TC e MID. As mensagens HELLO são responsáveis pelo processo de descoberta de vizinhos a um ou dois saltos de distância. Estas mensagens são geradas regularmente e enviadas em *broadcast* pelo emissor a fim de atualizar os valores dos enlaces para seus vizinhos.

Como mensagens HELLO são responsáveis por definir topologias locais, o protocolo OLSR define mensagens TC (*Topology Control*) para serem responsáveis

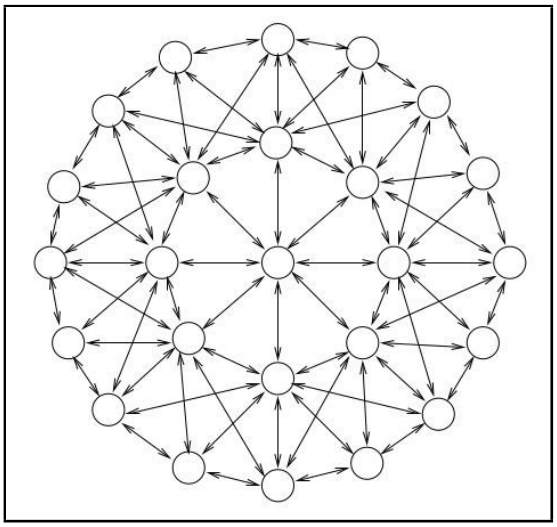

Figura 3 – Difusão de mensagens de controle pela rede utilizando a abordagem tradicional
Fonte: Adaptado de Tonnesen (2004)

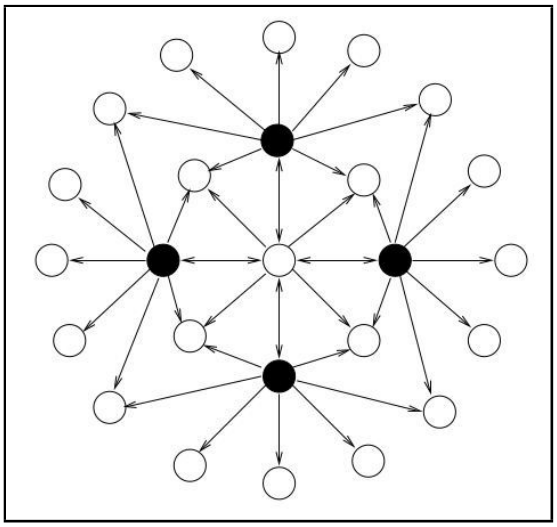

Figura 4 – Difusão de mensagens de controle utilizando a abordagem MPR
Fonte: Adaptado de Tonnesen (2004)

por difundir informações topológicas do roteador para toda a rede através dos nós MPR. A mensagem TC divulga o conjunto de endereços dos vizinhos que formam enlaces com o emissor da mensagem.

Mensagens MID (*Multiple Interface Declaration*) são enviadas apenas por roteadores com múltiplas interfaces OLSR. Alguns roteadores pertencentes à rede podem possuir múltiplas interfaces de comunicação, cada uma exigindo um identificador na rede (IP). Basicamente uma interface de comunicação é a porta de entrada e saída dos pacotes de dados do roteador, pacotes estes, endereçados a portas específicas. Mensagens MID são basicamente uma lista de endereços IPs associados às interfaces do roteador (TONNESEN, 2004).

Além de detectar a existência de enlaces entre os roteadores, as mensagens HELLO também são responsáveis por difundir a qualidade da conexão entre eles. A atribuição da qualidade do enlace é baseada na métrica ETX (*Expected Transmission Count*). A métrica ETX, apresentada por Couto (2004), propõe uma estimativa para qualidade de enlaces de redes sem fio baseada no número de transmissões de pacotes (incluindo retransmissões) necessárias para que um pacote seja enviado com sucesso.

A métrica ETX calcula seu coeficiente com base na relação entre o total de pacotes enviados e o número de pacotes entregues com sucesso. Valores do coeficiente próximos a 1 indicam 100% de eficiência nas transmissões e, a medida em que este coeficiente aumenta, maior o número de pacotes a serem enviados para se obter um caso de sucesso. Na prática, caminhos com as menores estimativas de ETX possuem as maiores taxas de transferência.

2.3 EMULADORES DE REDES

Devido às limitações relacionadas à disponibilidade de tempo e equipamentos, algumas técnicas e ferramentas foram desenvolvidas a fim de facilitar o processo de teste e validação de modelos e aplicações no contexto de redes.

Simuladores procuram representar uma rede através de modelos matemáticos de estruturas de dados e protocolos. Tais ferramentas visam disponibilizar um ambiente controlado e livre de interferências como alternativa ao ambiente real encontrado.

Emuladores de rede, por outro lado, não alteram o modelo aplicado em condições reais, apenas oferecem um ambiente compatível com o esperado pela aplicação, permitindo assim, que ela seja executada de forma similar à encontrada na prática.

Como os emuladores permitem que algoritmos e aplicações sejam expostos a condições similares às reais encontradas em redes sem fio, as conclusões obtidas a partir de resultados produzidos por um processo de emulação serão um grande indicativo dos prováveis resultados a serem encontrados em ambientes reais.

2.3.1 Olsr_switch

O OLSRD possui uma ferramenta de emulação chamada Olsr_switch (TONNESEN, 2005). A ferramenta permite executar instâncias do OLSRD em uma rede emulada. Cada instância emula o comportamento de um AP *mesh*, se comunicando com a rede através do protocolo TCP (CERF; KHAN, 1974), um protocolo para transporte de dados com um controle acerca da entrega dos pacotes.

O Olsr_switch gerencia dois conjuntos de dados, clientes e enlaces. Um cliente (instância do protocolo) é um elemento (nó) da rede que pode possuir enlaces unidirecionais para os demais APs da rede. Cada cliente encaminha seus pacotes para todos os APs com os quais ele possui um enlace válido.

Os enlaces podem ser manipulados de forma independente, podendo assumir três diferentes estados de acordo com a porcentagem de sucesso na entrega dos pacotes através daquela conexão:

- Aberto: todo pacote encaminhado será entregue com sucesso através deste enlace (qualidade 100%);
- Fechado: nenhum pacote será encaminhado com sucesso através deste enlace (qualidade 0%);
- Com perdas: a qualidade da conexão é definida pela taxa de sucesso na

entrega dos pacotes (qualidade [1, 99]%).

3 ALGORITMOS E ESTRUTURAS DE DADOS

A compreenção do funcionamento das filosofias de roteamento utilizadas pelo protocolo OLSR permite um visão geral e de alto nível acerca do funcionamento do protocolo.

Para um aprofundamento no processo de cálculo de caminhos mínimos do OLSRD é necessário o entendimento do funcionamento interno do protocolo. Para a compreenção deste processo faz-se necessário o entendimento de como as estruturas de dados utilizadas durante o cálculo de caminhos mínimos do protocolo estão organizadas.

Este capítulo apresenta uma visão geral acerca das estruturas de dados utilizadas pelo processo de cálculo de caminhos mínimos no protocolo OLSRD.

A Seção 3.1 descreve conceitos básicos de análise de algoritmos úteis para o entendimento e comparação de desempenho das estruturas de dados citadas neste trabalho.

A Seção 3.3 descreve o funcionamento do Algoritmo de Dijkstra, responsável pelo cálculo dos caminhos mínimos do protocolo e a Seção 3.2 descreve os conceitos básicos relacioados a filas de prioriodade, estrutura auxiliar utilizada pelo Algoritmo de Dijkstra.

As Seções 3.4, 3.5 e 3.6 descrevem o funcionamento das estruturas de dados utilizadas como filas de prioriodade para o Algoritmo de Dijkstra neste estudo, respectivamente o heap binário, a árvore AVL e a Árvore Van Emde Boas.

Com base nas seções apresentadas, pode-se descrever na Seção 3.7 a organização das estruturas do OLSRD relacionadas a representação da topologia da rede e a organização do Algoritmo de Dijkstra, ambas utilizadas no cálculo de caminhos mínimos do protocolo.

3.1 ANÁLISE DE ALGORITMOS

A análise de algoritmos visa, segundo Sedgewick e Flajolet (2013), estimar de forma confiável os recursos necessários para um algoritmo resolver as instâncias de um problema computável, permitindo avaliações acerca de sua eficiência de acordo com sua aplicação, além de comparações perante outros algoritmos de mesmo propósito.

Os recursos computacionais geralmente avaliados pela análise de algoritmos dizem respeito às estimativas de consumo de tempo e espaço pelo referido algoritmo,

ou seja, estimar por quanto tempo o algoritmo precisará ser executado para resolver o problema considerando todas as entradas válidas e a quantidade de memória utilizada pelo algoritmo neste processo.

Estimar os recursos requeridos pelo algoritmo independente da máquina na qual o algoritmo analisado será executado é uma restrição que a análise de algoritmos impõe para garantir a generalização dos resultados obtidos.

Ainda segundo Sedgewick e Flajolet (2013, p. 4), "a qualidade da implementação, propriedades do compilador, arquitetura da máquina e outras facetas do ambiente de programação têm efeitos dramáticos na performance do algoritmo", mas ainda assim, o desempenho segue um padrão esperado para o algoritmo com estas diferentes configurações. Na prática, a análise de algoritmos contabiliza apenas as operações inerentes ao próprio algoritmo, ignorando estes fatores externos na avaliação de desempenho. Desta forma, a análise de algoritmos visa encontrar um padrão matemático que descreva o desempenho do algoritmo.

3.1.1 Análise Assintótica

A contabilização das operações relevantes ao algoritmo durante sua execução resulta em uma função matemática f denominada *função de complexidade*, responsável por descrever a quantidade de recursos computacionais de tempo ou espaço necessários ao algoritmo. Em uma análise de complexidade de tempo, por exemplo, para a entrada de uma instância de tamanho n, um determinado algoritmo realiza uma quantidade de operações $f(n) > 0$. Exemplos de funções de complexidade podem ser vistos no Quadro 1.

Tabela 1 – Exemplos de funções de complexidade

Descrição	Função
Constante	1
Logarítmica	$\log n$
Linear	n
Linearítmica	$n \log n$
Quadrática	n^2
Cúbica	n^3
Exponencial	2^n

Fonte: Adaptado de Sedgewick e Wayne (2011)

A partir da função de complexidade, pode-se definir o quão eficiente um algoritmo é em relação a outro de mesmo propósito. Denotando $f(n)$ e $g(n)$ como as respectivas funções não-negativas de complexidade de dois algoritmos dados, as seguintes notações assintóticas são convencionadas na literatura relacionada:

- $f(n) \in O(g(n))$: $f(n)$ possui uma ordem de crescimento igual ou inferior à ordem de crescimento de $g(n)$. Diz-se que $f(n)$ é assintoticamente igual ou inferior à $g(n)$, se existem constantes, $c > 0$ e $n_0 \geq 0$, tal que $f(n) \leq c \cdot g(n), \forall n \geq n_0$, ou ainda, diz-se que $g(n)$ é um limite superior para $f(n)$;

- $f(n) \in \Omega(g(n))$: $f(n)$ possui uma ordem de crescimento igual ou superior à ordem de crescimento de $g(n)$. Diz-se que $f(n)$ é assintoticamente igual ou superior à $g(n)$, se existem constantes, $c > 0$ e $n_0 \geq 0$, tal que $f(n) \geq c \cdot g(n), \forall n \geq n_0$, ou ainda, diz-se que $g(n)$ é um limite inferior para $f(n)$;

- $f(n) \in \Theta(f(n))$: $f(n)$ possui uma ordem de crescimento igual à ordem de crescimento de $g(n)$. Diz-se que $f(n)$ é assintoticamente igual à $g(n)$, se existem constantes, $c_0 > 0$, $c_1 > 0$ e $n_0 \geq 0$, que satisfazem $f(n) \geq c_0 \cdot g(n), \forall n \geq n_0$ e $f(n) \leq c_1 \cdot g(n), \forall n \geq n_0$, ou ainda, diz-se que $g(n)$ é um limite superior e inferior para $f(n)$;

- $f(n) \in o(g(n))$: $f(n)$ posui uma ordem de crescimento inferior à ordem de crescimento de $g(n)$. Diz-se que $f(n)$ é assintoticamente inferior à $g(n)$, se existem constantes, $c > 0$ e $n_0 \geq 0$, tal que $f(n) < c \cdot g(n), \forall n \geq n_0$, ou ainda, diz-se que $g(n)$ é um limite superior firme para $f(n)$;

- $f(n) \in \omega(f(n))$: $f(n)$ posui uma ordem de crescimento superior à ordem de crescimento de $g(n)$. Diz-se que $f(n)$ é assintoticamente superior a $g(n)$, se existem constantes $c > 0$ e $n_0 \geq 0$, tal que $f(n) > c \cdot g(n), \forall n \geq n_0$, ou ainda, diz-se que $g(n)$ é um limite inferior para $f(n)$.

Considerando que as funções de complexidade podem ser compostas por vários termos que descrevem as operações efetuadas pelo algoritmo, a leitura da função pode ficar comprometida devido à presença de termos de menor ordem. Segundo Sedgewick e Wayne (2011), considerando que as constantes multiplicadoras e os termos de menor ordem são dominados assintoticamente pelo termo de maior ordem, é comum classificar um algoritmo apenas pelo termo dominante de sua função de complexidade, ignorando os demais.

O termo de maior valor da função de complexidade define a classe assintótica do algoritmo, também conhecida como ordem de crescimento. Exemplos de funções de complexidade descritas em termos de sua classe assintótica podem ser vistos no Quadro 1.

Podem haver situações em que o trecho de código que domina assintoticamente os demais, ou seja, o trecho que define a classe assintótica do algoritmo, é acessado apenas em casos específicos durante a execução. Em tais situações, definir uma classe assintótica mais alta (pior) para um algoritmo devido à execução esporá-

Função	Ordem de crescimento
$n^3/6 - n^2/2 + n/3$	n^3
$n^2/2 - n/2$	n^2
$\log n + 1$	$\log n$
3	1

Quadro 1 – Exemplos de classificação de funções de complexidade
Fonte: Adaptado de Sedgewick e Wayne (2011)

dica de um trecho de código mais custoso acaba não refletindo de forma justa o real desempenho do algoritmo.

Para contornar este problema, pode-se apelar para a análise amortizada que, segundo Cormen *et al.* (2009), define a média das operações necessárias para a execução do algoritmo. A análise amortizada busca principalmente mostrar que o custo médio de uma sequência de operações é pequeno, mesmo que uma única operação pertencente à sequência apresente um custo mais alto. Vale salientar que a "a análise amortizada difere da análise de caso médio, em que a probabilidade não está envolvida; uma análise amortizada garante o desempenho médio de cada operação, no pior caso." (CORMEN *et al.*, 2009).

Funções de complexidade de algoritmos representados por complexidades amortizadas são assinaladas pelo simbolo $\sim$, como por exemplo $\widetilde{O}(n^2)$ e $\widetilde{O}(n \log n)$.

3.1.2 Análise de Casos

O total de operações executadas por um algoritmo depende geralmente não só do tamanho da instância do problema recebida como entrada, como também das características apresentadas por ela. Isto se deve pelo fato de alguns algoritmos se comportarem de forma diferente (mesmo considerando entradas de mesmo tamanho) de acordo com certas características apresentadas pelas instâncias, fazendo-o, em alguns casos, executar trechos de código mais custosos computacionalmente que outros.

Visto que um mesmo algoritmo pode apresentar funções de complexidade distintas de acordo com as características das instâncias do problema, cria-se uma maneira de definir os limites apresentados pelo algoritmo considerando todas as instâncias válidas do problema. As principais classificações de complexidades apresentadas por um algoritmo são:

- Melhor caso: apresenta a menor classe assintótica $f(n)$ possível para um algoritmo considerando todas as instâncias válidas do problema, ou seja, a melhor performance esperada para o algoritmo. Usualmente diz-se que $f(n)$

é um limite inferior para o algoritmo;

- Pior caso: apresenta a maior classe assintótica $f(n)$ possível para um algoritmo considerando todas as instâncias válidas do problema, ou seja, a pior performance esperada para o algoritmo. Usualmente diz-se que $f(n)$ é um limite superior para o algoritmo;
- Caso médio: apresenta uma classe assintótica $f(n)$ que caracteriza a complexidade média esperada para o algoritmo de acordo com a relação de uma distribuição de probabilidade relativa ao domínio das instâncias do problema.

3.2 FILAS DE PRIORIDADE

Muitos algoritmos necessitam processar um conjunto de itens em ordem, não necessariamente todos de uma vez, e seguindo um determinado critério para a escolha do item a ser processado de acordo com a aplicação. Diferentemente de uma fila comum, a fila de prioridade é uma estrutura de dados que atribui adicionalmente uma prioridade (denominada chave) a cada elemento do seu conjunto (chamado de conjunto Q).

As filas de prioridade organizam seu conjunto Q de forma a garantir que um elemento com maior prioridade seja selecionado antes de um elemento de menor prioridade. Sempre que requisitada, a fila remove o item de maior prioridade no conjunto, se reorganizando de forma a disponibilizar o próximo item de maior prioridade no conjunto Q de maneira eficiente na próxima requisição.

As filas de prioridade podem ser organizadas de forma a priorizar valores de chaves mais altos (filas de prioridade de máximo) ou mais baixos (filas de prioridade de mínimo).

Segundo Cormen *et al.* (2009), dentre outras operações, uma fila de prioridade de máximo deve admitir:

- Inserção (Q, x): insere o elemento x no conjunto Q;
- Máximo (Q): retorna o elemento com o maior valor da chave (maior prioridade da fila);
- ExtraiMáximo (Q): remove e retorna o elemento com o maior valor da chave (maior prioridade da fila);
- IncrementaChave (Q, x, k): altera a prioridade do elemento x para o valor de k a fim de aumentar sua prioridade na fila.

Alternativamente, uma fila de prioridade de mínimo suporta as operações inserção, mínimo, ExtraiMínimo e DecrementaChave.

3.3 ALGORITMO DE DIJKSTRA

O Algoritmo de Dijkstra foi desenvolvido, segundo Bondy e Murty (2008), baseado nos algoritmos de Prim e de busca em largura (BFS - *Breadth-First Search*), resultando na melhor solução conhecida para resolver o problema dos caminhos mínimos de origem única (*single source shortest path* - SSSP) assumindo arestas não negativas.

O problema SSSP define-se por, dado um grafo $G = (V, A)$ constituído pelos conjuntos V de vértices e A de arestas, encontrar o menor caminho entre um vértice de origem $s \in V$ e os demais vértices ($v \in V$) do grafo. O Algoritmo de Dijkstra assume que para toda aresta $(u, v) \in A$ seu custo, denotado por $w(u, v)$, é não-negativo.

O algoritmo baseia-se na análise de duas estimativas atribuídas a cada vértice $v \in V$, denominadas $d(v)$ e $\pi(v)$, onde $d(v)$ é o custo do menor caminho entre a origem e o vértice v; e $\pi(v)$ é o vértice antecessor à v neste mesmo caminho.

O Algoritmo de Dijkstra mantém um conjunto S de vértices que já possuem o caminho mínimo definido a partir de s e, a cada iteração, com base nas estimativas de custo dos vértices do conjunto Q (formado pelo subconjunto $V \setminus S$), busca utilizar um vértice v deste conjunto com o menor custo associado como integrante do caminho mínimo para seus adjacentes. O processo de relaxamento, como é conhecido, visa atualizar o caminho mínimo dos vértices do conjunto Q a partir de v que, ao fim da iteração, é adicionado ao conjunto S. Um esboço do Algoritmo de Dijkstra pode ser visto no Quadro 2.

```
1  Dijkstra(G, w, s)
2      inicializa_estimativas(G, s)
3      S = 0
4      Q = G.V
5      enquanto Q != 0
6          u = extrai_minimo(Q)
7          S = uniao(S, u)
8          para cada vertice v em G.adj[u]
9              se d(u) + w(u,v) < d(v)
10                 d(v) = d(u) + w(u,v)
11                 pi(v) = u
12                 decrementa_chave(Q, v, d(v))
```

Quadro 2 – Algoritmo de Dijkstra
Fonte: Adaptado de Cormen *et al.* (2009)

Dado um grafo G, o conjunto w de custos das arestas e uma origem s, o algoritmo inicia a execução efetuando o procedimento de inicialização das estimativas $d(v)$ e $\pi(v)$, para todo vértice v pertencente a V (linha 2).

Com exceção do vértice inicial s, todo vértice $v \in V$ recebe um valor simbólico para a estimativa $d(v) = \infty$ indicando que ainda não se conhece o custo do caminho entre a origem s e o referido vértice. O vértice s recebe o valor 0, representando o custo para se chegar à ele mesmo. O valor de $\pi(v)$, para cada vértice $v \in V$, também é inicializado com um valor simbólico, indicando que até aquele momento na execução do algoritmo não se conhece um caminho até v.

O conjunto S indicado na linha 3 armazena os vértices que já possuem o caminho mínimo definido. Inicialmente nenhum vértice possui tal atributo e, portanto, S é vazio.

O algoritmo também armazena um conjunto Q representado por uma fila de prioridades que mantém todos os vértices que ainda não possuem seu custo mínimo definido. Sendo V o conjunto de todos os vértices do grafo, tem-se $Q = V \setminus S$.

O algoritmo procura calcular o custo do caminho mínimo para os vértices com base no vértice $u \in Q$ que possui a menor estimativa $d(u)$ a cada iteração do laço (linha 5), e este passa a integrar o conjunto S (linha 7) pois, uma vez removido do conjunto Q (linha 6), a estimativa $d(u)$ do referido vértice se torna permanente até o fim da execução do algoritmo. Para detalhes da prova de corretude do algoritmo, consultar Cormen *et al.* (2009).

Para cada vértice u selecionado, o algoritmo procura atualizar a estimativa de seus vizinhos (linhas 8 a 11). Caso o custo do caminho mínimo entre s e u somado ao custo da aresta entre u e v compuserem um caminho mais eficiente que o atualmente computado para o vértice v, u passará a compor o caminho mínimo para v, atualizando as estimativas $d(v)$ e $\pi(v)$. Com a melhora do custo mínimo do vértice, sua prioridade é ser aumentada, delegando ao procedimento decrementa_chave (linha 12) atualizar a fila de prioriodade com a mudança na prioriodade do vértice.

A fila de prioridade Q é determinante para a complexidade do algoritmo através da chamada de suas funções Inserir (implícita na linha 4), ExtraiMínimo (linha 6) e DecrementaChave (implícita na linha 12). Como o algoritmo não adiciona nenhum vértice ao conjunto Q após a linha 3, cada vértice é extraído do conjunto e adicionado à S exatamente uma vez (CORMEN *et al.*, 2009), ou seja, o laço de repetição das linha 5 à 12 é iterado $|V|$ vezes e por consequência, a operação ExtraiMínimo é executada apenas uma vez para cada vértice.

Como, para cada vértice v adjacente a u, as linhas 8-12 são executadas apenas uma vez, nenhuma aresta (u, v) será explorada mais de uma vez durante a execução do algoritmo. Logo o laço de repetição da linha 8 será executado $|A|$ vezes,

culminando em $O(|A|)$ chamadas da função decrementa_chave no pior caso.

A definição da complexidade de tempo de execução no pior caso para o Algoritmo de Dijkstra depende da complexidade das operações executadas sobre a fila de prioridades, que por sua vez dependem da estrutura de dados utilizada a armazenar a fila Q. O Quadro 3 apresenta a complexidade no pior caso de operações básicas em um fila de prioridades com n itens, armazenadas em diferentes estruturas de dados.

Estrutura de dados	Extrai máximo	Decrementa chave
heap binário	$O(\log_2 n)$	$O(\log_2 n)$
heap binomial	$O(\log_2 n)$	$O(\log_2 n)$
run-relaxed heap	$O(\log_2 n)$	$O(1)$

Quadro 3 – Complexidade no pior caso de operações básicas em diferentes filas de prioridades
Fonte: Adaptado de Oberhauser e Simha (1995)

A partir da utilização de um heap binário tem-se o custo de $O(\log_2 |V|)$ para cada operação de ExtraiMínimo, totalizando um tempo proporcional a $O(V \log_2 |V|)$ para as $|V|$ chamadas a essa função. Cada chamada à DecrementaChave possui um custo de $O(\log_2 |V|)$. Como o algoritmo possui $|A|$ chamadas a essa função, totaliza-se um tempo de $O(|A| \log_2 |V|)$. A complexidade final apresentada pelo algoritmo é $O((|V| + |A|) \log_2 |V|) \in O(|A| \log_2 |V|)$, assumindo $|V| \in O(|A|)$.

A utilização de filas de prioridades eficientes, principalmente em relação à complexidade da operação DecrementaChave assumindo $|V| \in O(|A|)$ "foi motivada a partir da observação de que o Algoritmo de Dijkstra tipicamente executa mais chamadas à função DecrementaChave do que chamadas à ExtraiMínimo" (CORMEN *et al.*, 2009).

3.4 HEAP BINÁRIO

Um heap binário é uma estrutura de dados organizada como uma árvore binária quase completa porém, sendo implementado classicamente em um espaço contíguo de memória, geralmente um vetor. A árvore representada pelo heap estará totalmente preenchida pelo menos até seu penúltimo nível. Caso o número de chaves não seja suficiente para preencher todo o último nível, os elementos armazenados nas folhas serão dispostos da esquerda para a direita neste nível.

Com a ajuda dos índices do vetor utilizado pelo heap, cada elemento da estrutura pode utilizar-se da organização do heap como uma árvore quase completa para encontrar a um custo constante seus relativos pai e filhos esquerdo e direito, se existirem.

Tendo o índice 1 como o responsável por indicar o nó raiz do heap, pode-se

definir a posição dos nós relacionados à raiz, generalizando estas propriedades para todos os demais elementos do heap. Sendo i o índice de um nó qualquer da estrutura, pode-se definir os procedimentos:

- pai (i): retorna o índice do nó pai de i a partir da fórmula $\lfloor i/2 \rfloor$;
- filhoEsquerdo (i): retorna o índice do filho esquerdo de i a partir da fórmula $2 \cdot i$;
- filhoDireito (i): retorna o índice do filho direito de i a partir da fórmula $2 \cdot i + 1$.

Heaps binários podem ser organizados de duas formas: priorizando valores mais altos das chaves armazenadas (sendo classificados como heap de máximo) ou priorizando valores mais baixos (heap de mínimo).

Diferentemente de uma árvore binária de pesquisa onde as chaves de menor valor ficam armazenadas mais à esquerda da estrutura, em um heap binário de mínimo, por exemplo, chaves de menor valor são salvas mais próximas à raiz da estrutura, independente do lado em que estejam armazenadas.

Esta característica apresentada pelo heap binário deve-se ao seu critério de classificação. Para um heap de mínimo, por exemplo, define-se que, para todo nó do heap, seus filhos (direito e esquerdo) terão chaves maiores que o nó do pai.

A Figura 5 ilustra uma árvore binária disposta como um heap de mínimo. Já a Figura 6 mostra a representação da árvore ilustrada na Figura 5 implementada na forma clássica de um vetor.

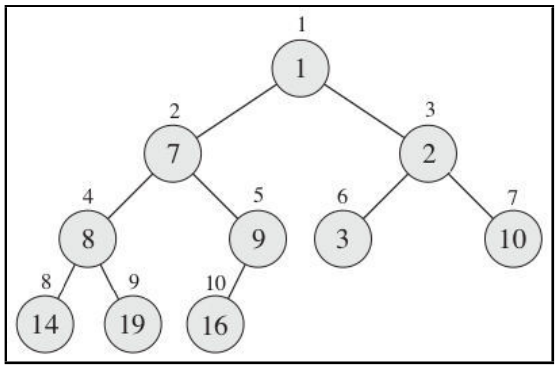

Figura 5 – Exemplo de um heap de mínimo ilustrado como uma árvore binária
Fonte: Cormen *et al.* (2009)

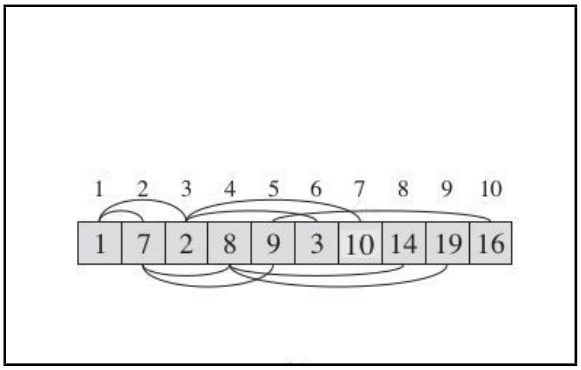

Figura 6 – Exemplo de um heap de mínimo ilustrado como um vetor
Fonte: Cormen *et al.* (2009)

Como um heap binário se mostra compatível com uma árvore binária completa, suas principais operações de fila de prioridade serão executadas na ordem de $O(\log_2(n))$, visto que a altura máxima de uma árvore completa é dada por esta função e as operações do heap baseiam-se na troca de elementos entre os níveis da árvore.

O procedimento de inserção em um heap binário se dá pela inserção do novo nó na última posição vazia do vetor, sendo necessário em seguida identificar a posição correta deste nó na estrutura, de acordo com sua chave perante seus nós antecessores a caminho da raiz. Este processo preenche o heap de cima para baixo e da esquerda para a direita, partindo inicialmente da raiz e inserindo o novo elemento na posição vazia mais à esquerda do último nível do heap. Este critério garante o preenchimento total do último nível da estrutura antes de se necessitar um novo nível. A Figura 7 ilustra o processo de inserção dos nós 9, 7, 2 e 10 em um heap binário. Nota-se o processo de correção da posição dos nós inseridos nas figuras (c) e (e).

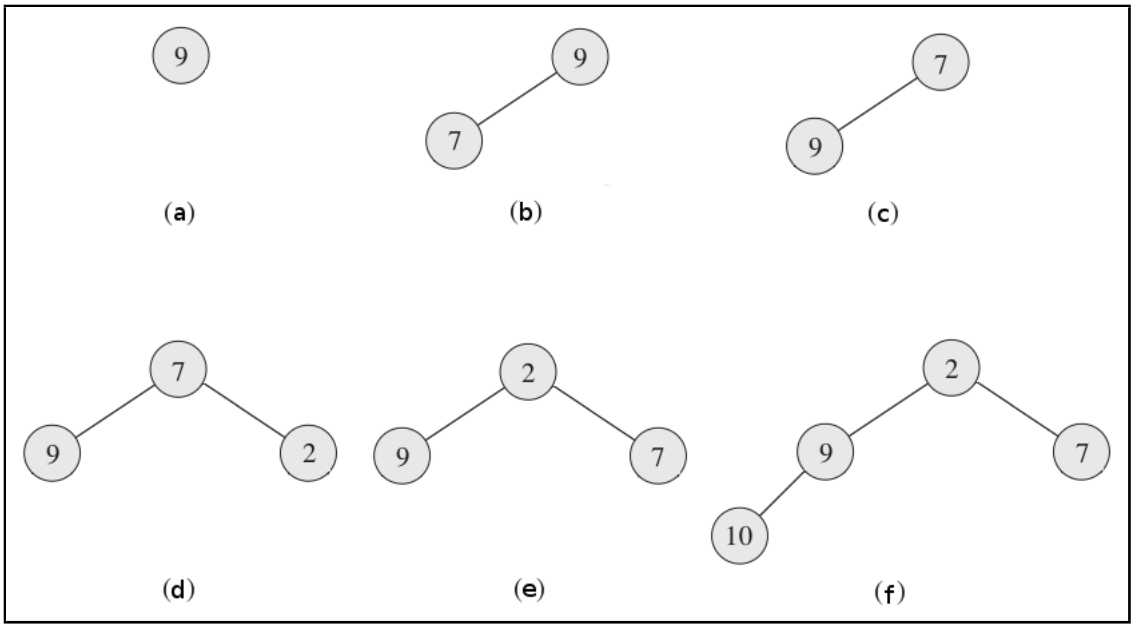

Figura 7 – Exemplo de inserções em um heap binário.
Fonte: Adaptado de Cormen *et al.* (2009)

O Quadro 4 apresenta um exemplo de pseudocódigo do procedimento de inserção. O procedimento possui como parâmetros a estrutura do heap binário e o valor da chave a ser inserida. As linhas 2 e 3 demostram a inserção de um novo elemento na última posição livre do vetor e com o auxílio do procedimento heap_decrementa_chave, a posição correta do elemento é calculada dentro da estrutura.

Inicialmente o item inserido recebe um valor simbólico alto para que durante a execução do procedimento heap_decrementa_chave a prioriodade do item seja atualizada com o valor de sua chave. O procedimento heap_decrementa_chave recebe a posição do elemento inserido e o valor a ser associado a ele. O uso deste procedimento auxiliar visa simplificar a construção do procedimento.

O procedimento DecrementaChave é responsável por atualizar para um valor menor a chave de um elemento da fila de prioridade, mas não necessariamente diminuindo sua prioridade na estrutura. Em um heap de mínimo, como usado no Algoritmo de Dijkstra, o procedimento DecrementaChave aumenta a prioridade de um elemento,

```
1 heap_insercao(Q, chave)
2     Q.tamanho = Q.tamanho + 1
3     Q.vetor[Q.tamanho] = INFINITO
4     heap_decrementa_chave(Q, Q.tamanho, chave)
```

Quadro 4 – Pseudocódigo do procedimento de inserção em um heap binário. Fonte: Adaptado de Cormen *et al.* (2009)

e o procedimento IncrementaChave, diminui.

O processo de atualização das estimativas de custo dos vizinhos de cada vértice durante a execução do Algoritmo de Dijkstra implica em sucessivas chamadas ao procedimento DecrementaChave a fim de manter a fila de prioridade atualizada com os custos (prioridades) dos vértices a cada iteração do algoritmo.

Em um heap binário, o procedimento DecrementaChave deve receber como parâmetro o índice do elemento a ser atualizado e o novo valor associado a ele. A partir destes dados o procedimento deve atualizar o referido nó e encontrar a posição correta do elemento atualizado dentro da fila a fim de manter suas propriedades.

A Figura 8 ilustra as configurações de um heap binário de mínimo durante o processo para o aumento da prioridade de um elemento, passado de uma prioridade com valor igual a 15 para a nova com valor igual a 4. Após a atualização da prioridade do elemento (imagem (a)), o procedimento procura corrigir a organização do heap para manter sua propriedade. O elemento atualizado assume o lugar de seu pai caso seu novo valor o coloque em maior prioridade na fila (imagens (b) e (c)). Após a identificação da posição do elemento atualizado o procedimento se encerra, mantendo o heap organizado (imagem (d)).

O Quadro 5 apresenta um exemplo de pseudocódigo do procedimento DecrementaChave. A função DecrementaChave deve receber como parâmetro a fila de prioridade atual, o índice do elemento a ser atualizado e seu novo valor.

Assumindo que o procedimento irá diminuir o valor da chave do elemento i, um teste é efetuado na linha 2 para evitar a desorganização do heap caso o valor recebido não corresponda ao esperado pelo procedimento. Com base no valor recebido, a prioridade do elemento é atualizada (linha 5) e tem-se início o processo de reorganização do heap (linhas 6 a 8). Cada iteração do algoritmo visa verificar e trocar, se necessário, a posição do elemento atualizado com seu pai, visando manter os elementos com maior prioridade acima na ramificação do heap em direção à raiz da estrutura. Ao fim do procedimento o elemento atualizado estará na sua posição correta no heap.

Devido ao heap binário possuir no máximo $\log_2 n$ níveis de altura e ao fato de que, a cada iteração sobe-se um nível em direção a raiz com a troca entre os nós pai e filho, o procedimento possuirá no máximo $\log_2 n$ trocas efetuadas a um custo constate, resultando na complexidade final do procedimento de $O(\log_2 n)$

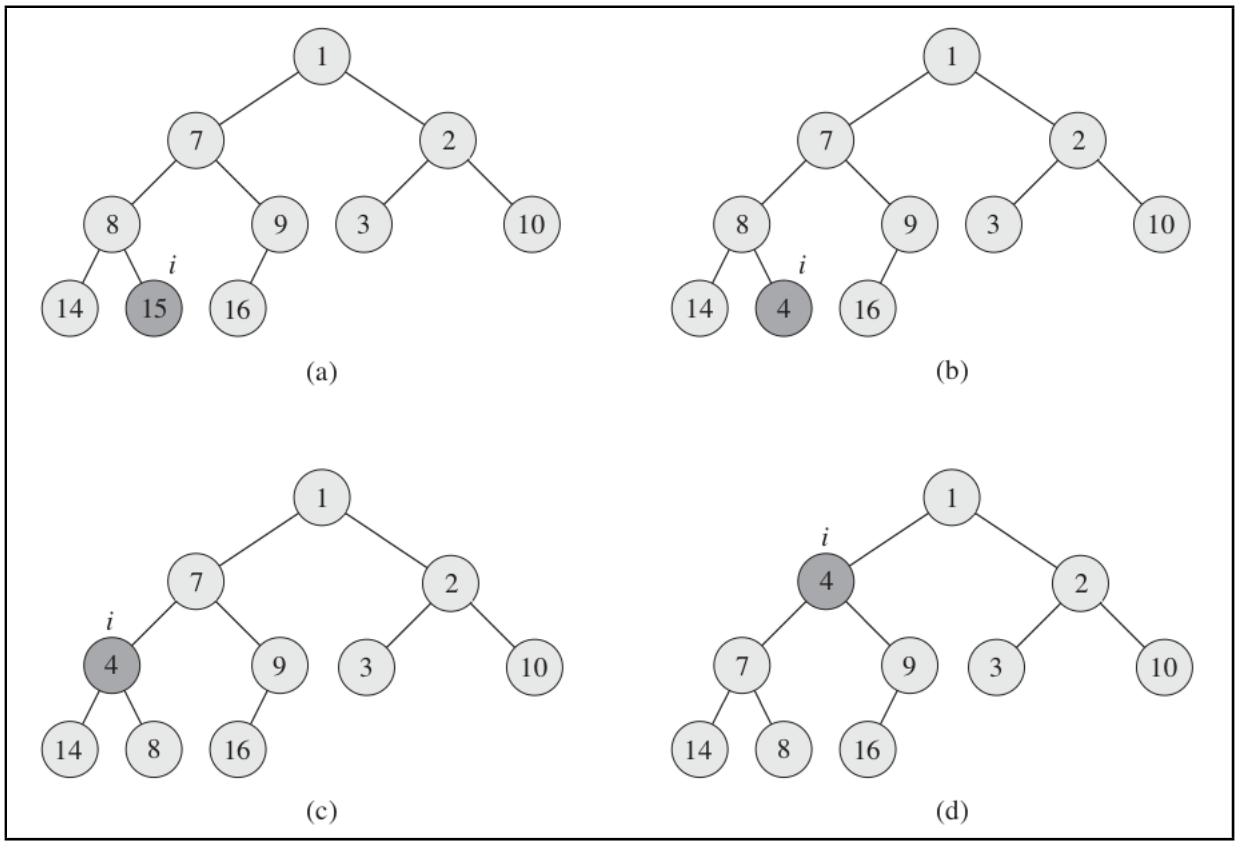

Figura 8 – Exemplo de uma operação DecrementaChave atualizando o valor da chave 15 para 4, aumentando sua prioridade na fila.
Fonte: Adaptado de Cormen *et al.* (2009)

```
heap_decrementa_chave(Q, i, chave)
    se(chave > Q.vetor[i]
        retorne "Erro – nova chave nao aumenta a prioridade do elemento Q.vetor[i]"
    Q.vetor[i] = chave
    enquanto i > 1 e Q.vetor[pai(i)] < Q.vetor[i]
        troca_pai_com_filho(Q.vetor[i], Q.vetor[pai(i)]
        i = pai(i)
```

Quadro 5 – Pseudocódigo do procedimento para aumento de prioridade de um elemento em um heap binário de mínimo
Fonte: Adaptado de Cormen *et al.* (2009)

A cada iteração do Algoritmo de Dijsktra, o vértice com a menor estimativa de custo mínimo é selecionado para ser usado como uma tentativa de compor o caminho mínimo entre seus vértices adjacentes pertencentes a $V \setminus S$ e o vértice de origem do grafo. Devido ao fato do heap binário de mínimo armazenar os nós com os menores valores chave mais próximos da raiz, o elemento de menor chave é o próprio nó raiz. Para identificar o vértice a ser explorado a cada iteração, basta selecionar o elemento situado na primeira posição do heap, como ilustrado no Quadro 6. Como o procedimento realiza apenas uma consulta ao primeiro elemento da estrutura, a complexidade

da função está na ordem de $\Theta(1)$.

```
1 heap_minimo(Q)
2     retorne Q.vetor[1]
```

Quadro 6 – Pseudocódigo do procedimento para identificação do elemento de maior prioridade de um heap binário de mínimo
Fonte: Adaptado de Cormen *et al.* (2009)

Devido a condição de parada do Algoritmo de Dijkstra estar relacionada ao fim da existência de vértices a serem explorados, estes armazenados no conjunto Q, ou seja, na fila de prioridade, há a necessidade de se extrair os vértices já explorados da estrutura. A cada iteração o elemento de menor custo é removido da fila de prioridade e movido para o conjunto de vértices já explorados (S). O procedimento responsável pela extração deste elemento da fila é o ExtraiMínimo.

Em um heap binário de mínimo, o procedimento ExtraiMínimo remove o nó raiz do heap e move o elemento da última posição do vetor para o a posição vaga da raiz. Como um elemento é removido da estrutura, uma posição deve se tornar vazia na fila. Devido a inserção de um novo elemento ser feita sempre na última posição do heap, ao remover um elemento, esta posição deve se tornar vaga. Devido ao novo elemento raiz ter sido um elemento do último nível do heap, sua prioridade não é suficiente para lhe garantir uma posição na raiz da estrutura, sendo necessário agora uma reorganização no heap.

A reorganização do heap se dá pelo posiocionamento correto do novo elemento raiz na estrutura. A cada iteração o elemento substituto desce um nível no heap até que sua posição satisfaça a propriedade de comparação entre as prioridades do elemento e seu relativos filhos. A Figura 9 apresenta um exemplo de execução do procedimento ExtraiMínimo, movendo o elemento de prioridade igual a 16 para a posição do nó raiz quando esta fica vaga. Após o deslocamento do nó, dá-se início ao processo de correção para encontrar a posição correta do elemento com uma prioriodade igual a 16 dentro do heap.

O Quadro 7 apresenta um exemplo de pseudocódigo do procedimento ExtraiMínimo de um heap binário de mínimo. A linha 2 possui um teste para informar se a fila de prioridade esta vazia, caso contrário o processo de remoção do nó raiz é iniciado. A linha 4 ilustra o procedimento que extrai o nó raiz do heap e nas linhas 5 e 6 a inserção do último elemento da fila na posição vaga e a atualização do número de elementos contido do heap, respectivamente. A partir da linha 7 inicia-se o processo de atualização do heap de acordo com o valor da nova raiz. Enquanto o valor no elemento substituto, identificado pelo índice i, possuir um valor maior que algum dos seus filhos (linha 9), a troca com o filho de maior prioridade será efetuada (linhas 10, 11 e 12). Ao se corrigir a posição do elemento substituto, o procedimento retorna o nó raiz

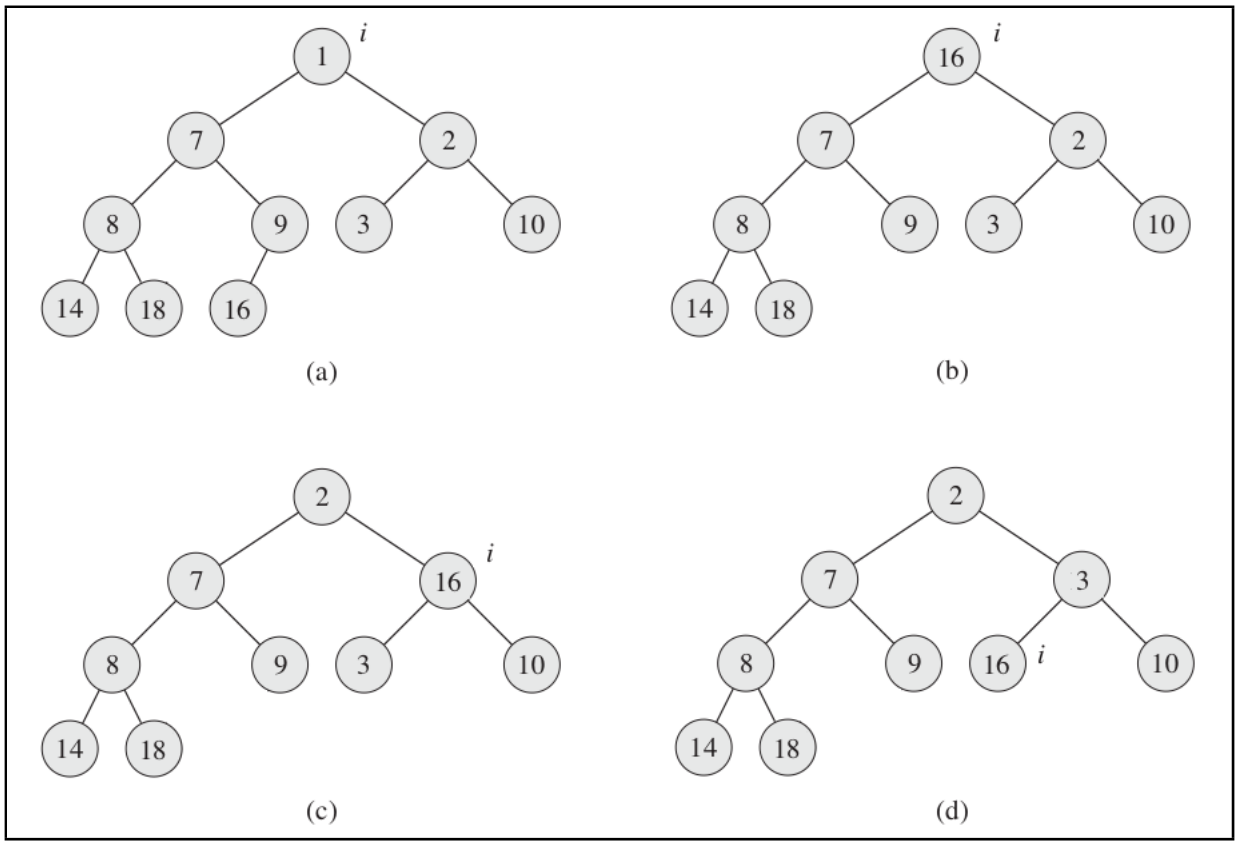

Figura 9 – Exemplo de uma remoção do elemento de maior prioridade do heap.
Fonte: Adaptado de Cormen *et al.* (2009)

original para a função que o invocou.

```
heap_extrai_minimo(Q)
    se(Q.tamanho = 0)
        retorne "heap vazio"
    min = Q.vetor[1]
    Q.vetor[1] = Q.vetor[Q.tamanho]
    Q.tamanho = Q.tamanho - 1
    i = 1
    j = filho_de_maior_prioridade(filho_direito(Q.vetor[i]),
   filho_esquerdo(Q.vetor[i]))
    enquanto(Q.vetor[i] > Q.vetor[j])
        troca_pai_com_filho(Q.vetor[i], Q.vetor[j])
        i = j
        j = filho_de_maior_prioridade(filho_direito(Q.vetor[i]),
   filho_esquerdo(Q.vetor[i]))
    retorne min
```

Quadro 7 – Pseudocódigo do procedimento ExtraiMínimo de um heap binário
Fonte: Adaptado de Cormen *et al.* (2009)

3.5 ÁRVORES BINÁRIAS

Árvores binárias e em particular árvores binárias de pesquisa (*Binary Search Tree* - BST) são estruturas de dados que têm como objetivo organizar rapidamente seus dados de forma dinâmica. Basicamente a estrutura organiza cada um de seus dados (nós) de forma ordenada a fim de se conseguir uma rápida busca posteriormente.

A partir de um nó inicial (raiz), ao inserir e remover itens da estrutura, a árvore binária organiza seu *layout* de acordo com o valor (chave) associado a cada item. Cada nó da árvore pode possuir subárvores associadas aos seus filhos (esquerdo e direito). Nós que não possuem subárvores associadas a seus filhos são chamados de nós folhas.

O critério adotado na organização de uma árvore binária impõe que todos os nós da subárvore associada ao filho esquerdo de um nó com chave x tenha uma chave com valor inferior a x; análogamente, todos os nós da subárvore associada ao filho direito do nó x devem possuir chaves com valores superiores a x.

Ao organizar sua estrutura de forma ordenada, a árvore pode-se valer dos benefícios assintóticos de uma busca binária onde, por exemplo, dada a busca por uma chave com valor superior ao da chave do nó raiz, garante-se que se existir um item com tal chave, esta se encontrará na subárvore associada ao filho direito do nó raiz. Essa organização elimina a necessidade de busca pela chave em toda a subárvore associada ao filho esquerdo do nó raiz e vice-versa. A Figura 10 apresenta um exemplo de uma árvore binária de pesquisa.

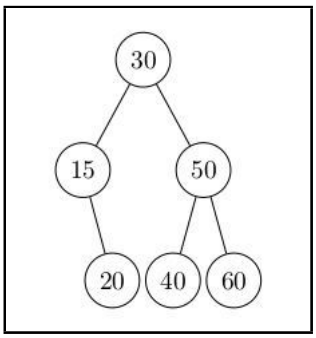

Figura 10 – Exemplo de uma árvore binária de pesquisa.
Fonte: Autoria própria

Apesar de a altura da árvore tender à $\log_2(n)$ levando a BST a possuir uma complexidade média para suas operações de $O\log_2(n)$, a implementação clássica de uma BST não prevê um tratamento para situações onde a árvore se torna degenerada, ou seja, apesar de ainda manter o critério de organização, a sequência de valores inseridos levam o processo de busca a um desempenho desinteressante ($O(n)$), como

se pode observar na Figura 11.

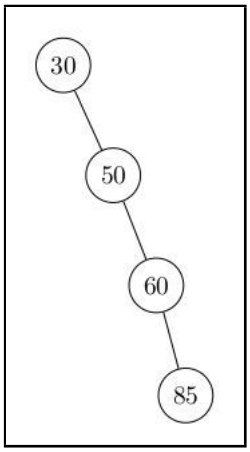

Figura 11 – Exemplo de uma árvore binária de pesquisa degenerada.
Fonte: Autoria própria

3.5.1 Árvore AVL

A fim de solucionar este problema, Adelson-Velsky e Landis (1962) propuseram um novo modelo de organização para as árvores binárias de pesquisa no qual a estrutura permite um processo de autobalanceamento. As árvore AVL, como ficaram conhecidas, tem como critério de balanceamento a divergência entre as alturas das subárvores esquerda e direita de cada nó. O balanceamento de uma árvore AVL garante à estrutura, para uma quantidade n de elementos, uma altura máxima de $1.4405 \cdot \log_2(n+2) - 0.3277$ (KNUTH, 1998), ou simplificando, $O(\log_2 n)$.

Para estimar o nível de balanceamento da árvore, cada nó passa a ter um valor inteiro adicional associado a ele, o fator de balanceamento, responsável por indicar a diferença entre as alturas das suas subárvores. A árvore é considerada balanceada se o valor indicado pelo fator de balanceamento é inferior a 1. O valor 0 para o fator balanceamento indica que ambas as subárvores possuem a mesma altura. Valores negativos indicam que a subárvore esquerda é maior que a direita enquanto que valores positivos indicam o contrário. Valores entre -1 e 1 são tolerados pela estrutura, porém diferenças maiores que estas necessitam da aplicação de um procedimento de rebalanceamento, executado por meio de rotações entre as posições dos nós envolvidos.

Uma árvore AVL permite dois tipos de rotações, dependendo da configuração da árvore após seu desbalanceamento, tanto para o lado esquerdo quanto para o lado direito, sendo elas as rotações simples e as rotações duplas.

Uma rotação simples ocorre quando o filho da maior subárvore de um nó desbalanceado (esquerda para fator de balanceamento igual a -2 e direita quando

o fator de balanceamento é igual a 2) possuir um fator de balanceamento neutro (0) ou no máximo uma leve inclinação para o mesmo valor do fator de balanceamento do nó pai (pai igual a 2 e filho igual à 1 ou pai igual a -2 e filho igual a -1). A Figura 12 apresenta um exemplo de rotação simples à esquerda, onde os nós envolvidos, 50 e 60, possuem uma inclinação para o lado direito (fator de balanceamento 2 e 1 respectivamente).

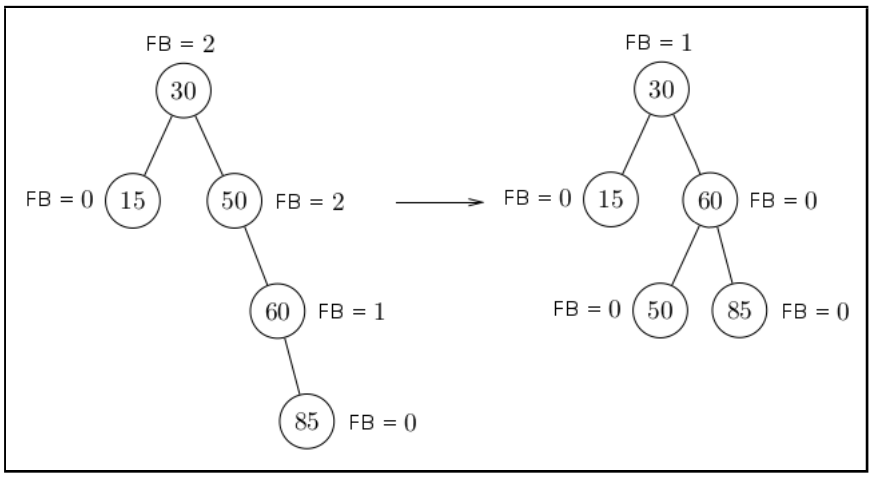

Figura 12 – Exemplo de uma rotação simples à esquerda entre os nós 50 e 60
Fonte: Autoria própria

Rotações duplas ocorrem quando o filho da maior subárvore de um nó desbalanceado possui uma inclinação para o lado contrário ao nó. Como por exemplo o nó pai com fator de balanceamento igual a 2 e o filho com fator de balanceamento igual a -1. A Figura 13 ilustra o processo de uma rotação dupla à esquerda.

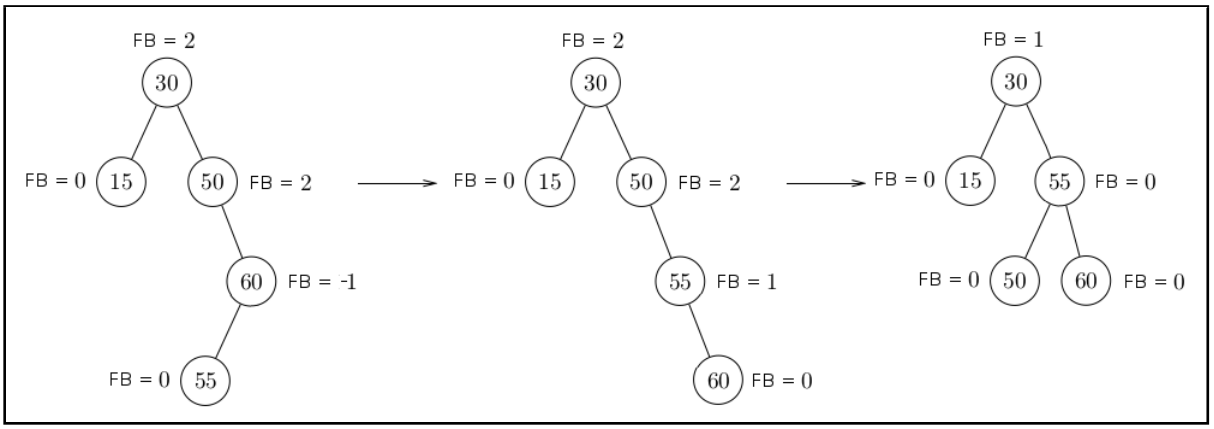

Figura 13 – Exemplo de uma rotação dupla à esquerda entre os nós 50 e 60
Fonte: Autoria própria

As operações de rotação podem ser executadas em tempo $\Theta(1)$, incluindo a atualização dos fatores de balanceamento, permitindo que a árvore AVL mantenha uma complexidade, no pior caso, na ordem de $O(\log_2(n))$ para suas operações de consulta, inserção e remoção.

A inserção em uma árvore AVL se dá inicialmente pela identificação do local em que deve ser inserido o nó. Como a AVL também é uma árvore binária de pesquisa, a busca por uma chave será executada em no máximo $\log_2 n$ operações. Definida a posição do elemento na árvore, o mesmo é inserido no local, realizando posteriormente a atualização dos fatores de balanceamento dos nós envolvidos na operação. Caso seja necessário, o rebalanceamento da árvore será executado, sendo as operações de atualização e rebalanceamento executadas em um tempo constante.

O procedimento de remoção de uma chave na árvore é executado de forma similar ao procedimento de inserção. A chave é pesquisada em um tempo $\log_2 n$ na estrutura, removida de seu lugar atualizando a relação entre os nós envolvidos na operação. Com a nova configuração da árvore os fatores de balanceamento são ajustados e se necessário o rebalanceamento é executado.

Apesar dos procedimentos de atualização e rebalanceamento serem executados em um tempo contante, assim como na inserção, estes passos podem ser executados mais de uma vez, uma para cada nível da árvore, o que não ocorre na inserção. Apesar de serem executados mais vezes a complexidade do procedimento de inserção mantem-se na ordem de $O(\log_2 n)$.

3.5.2 Threaded Binary Tree

Apesar de uma árvore AVL assegurar uma complexidade de $O(\log_2(n))$ para suas operações básicas, uma de suas operações, responsável por selecionar todos os seus nós em ordem crescente, denominada percurso inordem, acaba requerendo um espaço extra de memória de $O(\log_2(n))$ devido à sua pilha recursiva.

O percurso inordem consiste em recursivamente visitar primeiramente os elementos de menor valor na árvore (elementos da subárvore associada ao filho esquerdo de cada nó) para posteriormente visitar os elementos de maior valor (aqueles pertencentes à subárvore associada ao filho direito do nó). A Figura 14 apresenta um exemplo de percurso inordem resultando na visita aos nós na ordem A, B, C, D, E, F, G, H, I. Há também outras variações do procedimento (pré ordem e pós ordem) que utilizam outros critérios na escolha da sequência de nós a serem visitados.

Morris (1979) apresentou um modelo de árvore binária que permite um procedimento inordem iterativo (além de suas variações) sem a necessidade de um espaço extra de memória ou de modificar drasticamente a estrutura da BST original.

Inicialmente foram adicionados a cada nó x, dois novos campos responsáveis por indicar os nós imediatamente anterior e posterior a x, baseados na ordem crescente dos valores na árvores. Basicamente os campos apontavam o próximo nó

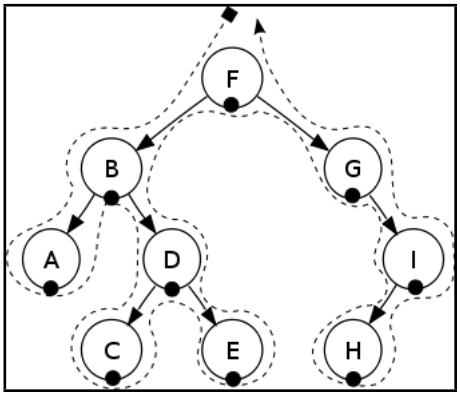

Figura 14 – Exemplo de execução do procedimento inordem
Fonte: Wikimedia Commons (2015)

que seria visitado pelo percurso inordem, ou seja, os dois nós com os valores mais próximos ao valor do nó x.

Partindo do nó de menor valor da árvore, pode-se "passear" pela árvore iterativamente apenas perguntando ao nó visitado qual o próximo nó da sequência. Como a atualização destes novos campos é feita em tempo constante durante os procedimentos de inserção e remoção da árvore, esta nova estrutura manteve a complexidade das operações de uma BST original.

3.6 ÁRVORE VAN EMDE BOAS

A árvore Van Emde Boas (BOAS, 1977), também conhecida como VEB, é uma estrutura de dados que armazena um conjunto de chaves, podendo assumir características comuns a outras estruturas, como vetores associativos e filas de prioridade, dependendo de sua implementação. Esta estrutura permite apenas a inserção de chaves inteiras que respeitam o intervalo $[0, u-1]$, onde u é a potência de 2 que representa o número de chaves válidas.

As operações suportadas por uma VEB bem como a explicação sobre seus funcionamentos descrita nesta seção são baseadas em Cormen *et al.* (2009). Dentre as operações suportadas, dada uma estrutura Q, que seja uma árvore Van Emde Boas, destacam-se:

- Inserção (Q, x): insere a chave x no conjunto Q;
- Remoção (Q, x): remove a chave x do conjunto Q;
- Busca (Q, x): retorna verdadeiro caso a chave x pertença ao conjunto Q, retorna falso caso contrário;
- Mínimo (Q): retorna o valor da menor chave do conjunto Q;

- Máximo (Q): retorna o valor da maior chave do conjunto Q;
- Predecessor (Q, x): retorna o valor da maior chave pertencente a Q que seja menor que a chave x, retorna nulo se não há elementos menores que x;
- Sucessor (Q, x): retorna o valor da menor chave pertencente ao conjunto Q que seja maior que a chave x, retorna nulo se não há elementos maiores que x.

A estrutura se divide recursivamente em $\sqrt{u}$ subconjuntos. Cada conjunto (também chamado de nó) de capacidade u é denominado por VEB(u) possui $\sqrt{u}$ ponteiros para subestruturas de tamanho $\sqrt{u}$ que repetem o processo até se obter conjuntos de capacidade igual a 2 (denotados por VEB(2)). Tal abordagem garante à estrutura uma complexidade de tempo na ordem de $O(\log_2 \log_2 u)$ para as operações de inserção, remoção, busca, predecessor e sucessor, além de executar as operações de mínimo e máximo com complexidade $\Theta(1)$.

Cada conjunto de uma VEB possui dois campos responsáveis por armazenar diretamente até dois valores chave distintos, indicando o menor e o maior valor chave armazenado no respectivo conjunto. O acesso a essas variáveis permite uma rápida definição do intervalo dos valores das chaves armazenadas no conjunto

O auxílio dos campos *min* e *max* permite saber em tempo $\Theta(1)$ se o nó VEB está vazio, possui exatamente uma ou pelo menos duas chaves. Caso ambos os campos *min* e *max* estejam vazios, o nó VEB não possui elementos. Caso ambos os campos sejam iguais, o nó VEB possuirá apenas uma chave e, caso ambos possuam valores distintos, o nó VEB estará armazenando duas ou mais chaves.

O Quadro 8 apresenta um exemplo de pseudocódigo para a função Mínimo, responsável por retornar o valor do campo min de um nó VEB e por consequência, o menor valor de todas as subárvores contidas no nó. Caso o nó seja a raiz da estrutura, o valor do campo *min* representará o menor valor chave de toda a árvore. Analogamente, o Quadro9 ilustra o pseudocódigo do procedimento Máximo.

```
1 veb_minimo(Q)
2     retorne Q.min
```

Quadro 8 – Exemplo de um pseudocódigo do procedimento Mínimo de uma VEB
Fonte: Adaptado de Cormen *et al.* (2009)

```
1 veb_maximo(Q)
2     retorne Q.max
```

Quadro 9 – Exemplo de um pseudocódigo do procedimento Máximo de uma VEB
Fonte: Adaptado de Cormen *et al.* (2009)

Como todo nó VEB de capacidade u, denotado por VEB(u) armazena chaves no intervalo $[0, u-1]$, as chaves que possuem valores maiores que o assinalado pelo campo *min* (incluindo a chave armazenada em *max*) deverão ser armazenadas em subestruturas de tamanho $\sqrt{u}$. Cada uma das $\sqrt{u}$ subestruturas armazena recursivamente chaves em um intervalo $[0, \sqrt{u}-1]$ em relação à estrutura pai, totalizando $\sqrt{u} \cdot \sqrt{u}$ valores, ou seja, todos os u valores que o conjunto VEB (u) armazena.

Cada nó x contém um vetor denominado *cluster*, e em cada posição do *cluster* há uma referência para uma subestrutura associada ao nó x. Os índices do vetor *cluster* que contém subestruturas não vazias são definidos a partir das chaves incluídas em x.

A localização do item do vetor *cluster* é definida de acordo com sua chave comparada ao intervalo de valores do *cluster*. Devido ao *cluster* referenciar $\sqrt{u}$ subconjuntos de tamanho $\sqrt{u}$, a posição da chave inserida é calculada para direcionar a chave a um dos $\sqrt{u}$ subconjuntos. O cálculo consiste basicamente na divisão do valor da chave pelo número de subconjuntos. O resultado aproximado para baixo indica a qual subconjunto a chave inserida pertencerá. A Fórmula (1) ilustra a função high(x) utilizada para o cálculo da posição da chave representada pela variável x nos subconjuntos referenciados por *cluster*.

$$high(x) = \lfloor x / \lfloor \sqrt{u} \rfloor \rfloor \tag{1}$$

$$low(x) = x mod \lfloor \sqrt{u} \rfloor \tag{2}$$

$$index(x, y) = x \cdot \lfloor \sqrt{u} \rfloor + y \tag{3}$$

Como o intervalo do subconjunto representa $\sqrt{u}$ do intervalo u do nó original, a escala da chave inserida também deve ser ajustada para se adequar ao novo intervalo, para isto o procedimento low(x) (visto na Fórmula (2)) calcula o novo valor da chave baseado no módulo da divisão entre a chave e o intervalo do subconjunto definido pela Fórmula (1).

Para otimizar a identificação de subconjuntos não vazios no *cluster* e facilitar a busca por chaves contidas nesses subconjuntos, cada nó VEB possui campo denominado *resumo*(summary) que faz referência a um subconjunto especial. Em um nó VEB do tipo *resumo*, as chaves armazenadas nesse subconjunto representam os índices preenchidos no vetor *cluster* de seu pai. A utilização do conjunto *resumo* permite a identificação de um item no vetor *cluster* em uma complexidade de tempo de $O(\log_2 \log_2(\sqrt{u}))$.

A utilização do vetor *cluster*, bem como do nó *resumo* se tornam desnecessários para nós VEB (2), visto que os campos *min* e *max* já satisfazem o papel de informar os valores de todas as chaves associadas ao nó, que neste caso são no máximo duas chaves. A Figura 15 ilustra a estrutura de um nó VEB(u) para $u > 2$.

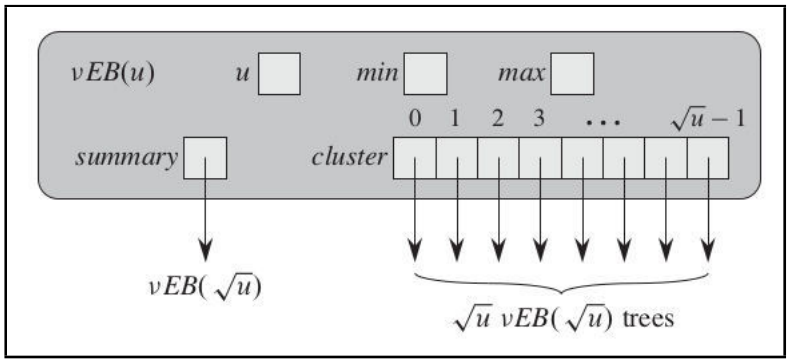

Figura 15 – Estrutura de um nó VEB.
Fonte: Cormen *et al.* (2009)

Para se utilizar uma árvore Van Emde Boas, o intervalo de valores válidos para u deve ser conhecido. A partir de tal valor pode-se criar o número necessário de nós VEB para armazenar os valores esperados, resultando em uma complexidade de espaço de $O(u)$.

A Figura 16 apresenta um exemplo de uma VEB(16) contendo o conjunto de chaves 2, 3, 4, 5, 7, 14 e 15. Como o menor valor inserido na árvore é a chave 2, o campo *min* do nó raiz armazena este valor bem como o campo *max* armazena o valor 15. Diferentemente do valor 15, porém, a chave 2 não é replicada nos subconjuntos do nó VEB(16). Para acomodar todas as chaves do nó no intervalo (*min*, *max*], a chave x é redirecionada para a posição high(x) do vetor *cluster*. O valor de cada chave x é recalculado pela função low(x) para atender o intervalo de valores desses nós. Os valores 0, 1 e 3 inseridos no elemento *resumo* VEB(4) indicam os índices do campo *cluster* de VEB(16) que possuem subconjuntos não vazios. Nota-se também um exemplo de conjunto com apenas uma chave, como é o caso do nó associado à posição 0 do campo *cluster* de VEB(16), onde apenas o valor 3 compõe o conjunto.

Possuir uma complexidade que independe do tamanho da entrada, mas sim de uma constante u que define o intervalo de valores aceitáveis pela estrutura pode garantir ganhos significativos de escalabilidade desde que o intervalo de valores seja relativamente pequeno.

A partir das operações suportadas pela VEB pode-se obter características de uma fila de prioridade considerando as equivalências apresentadas no Quadro 10.

O procedimento de inserção na árvore Van Emde Boas se dá inicialmente pelo preenchimento dos campos *min* e *max*. Inicialmente com a inserção da primeira chave, o intervalo de valores contido no nó é delimitado por um único elemento, atribuindo o mesmo valor da chave nos campos *min* e *max* do nó. A Figura 17 ilustra exemplos de inserções em uma árvore VEB(4), onde o processo inicial de inserção de uma chave pode ser visto na transição entre os itens (a) e (b). O Quadro 11 ilustra o pseudocódigo de inserção em um nó VEB previamente vazio.

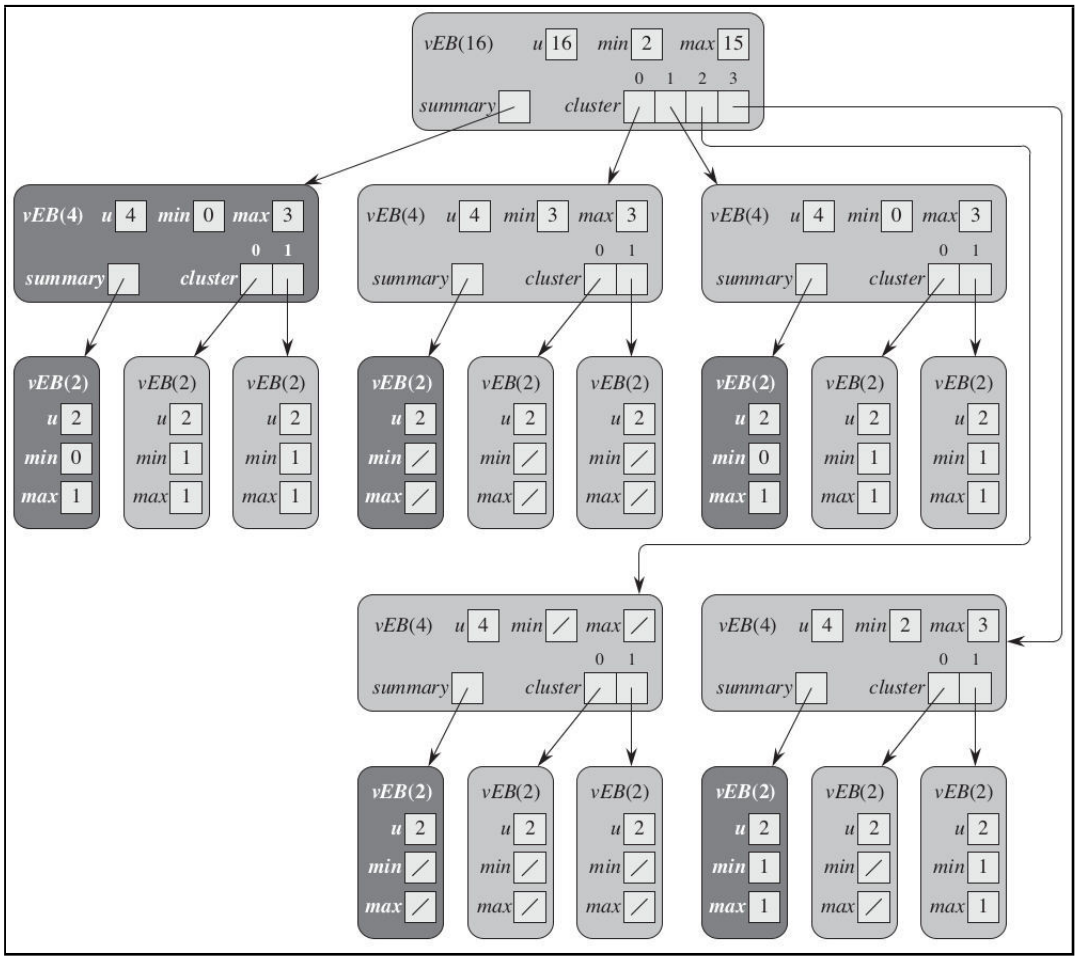

Figura 16 – Exemplo de uma árvore Van Emde Boas com *u* igual a 16
Fonte: Adaptado de Cormen *et al.* (2009)

Fila de prioridade	***VEB***
Operação	
Mínimo	Mínimo
Inserção	Inserção
ExtraiMínimo	Mínimo + Remoção
DecrementaChave	Membro + Remoção + Inserção

Quadro 10 – Equivalência entre as operações de filas de prioridade e as oferecidas pela árvore Van Emde Boas
Fonte: Autoria própria

A inserção do segundo elemento implica na atualização do limite inferior ou superior do nó (ilustrado na transição do item (b) para o item (c) na Figura 17). Caso a chave inserida seja inferior à já existente, esta chave passa a ser armazenada no campo *min*, mantendo a chave anterior no campo *max*, caso contrário, a nova chave é armazenada no campo *max*. Independente da localização do novo elemento, a partir da inserção de duas ou mais chaves, o nó raiz direciona os elementos maiores que o valor de *min* para uma subestrutura referenciada no vetor *cluster*.

A partir da criação de uma subestrutura no campo *cluster*, a atualização do

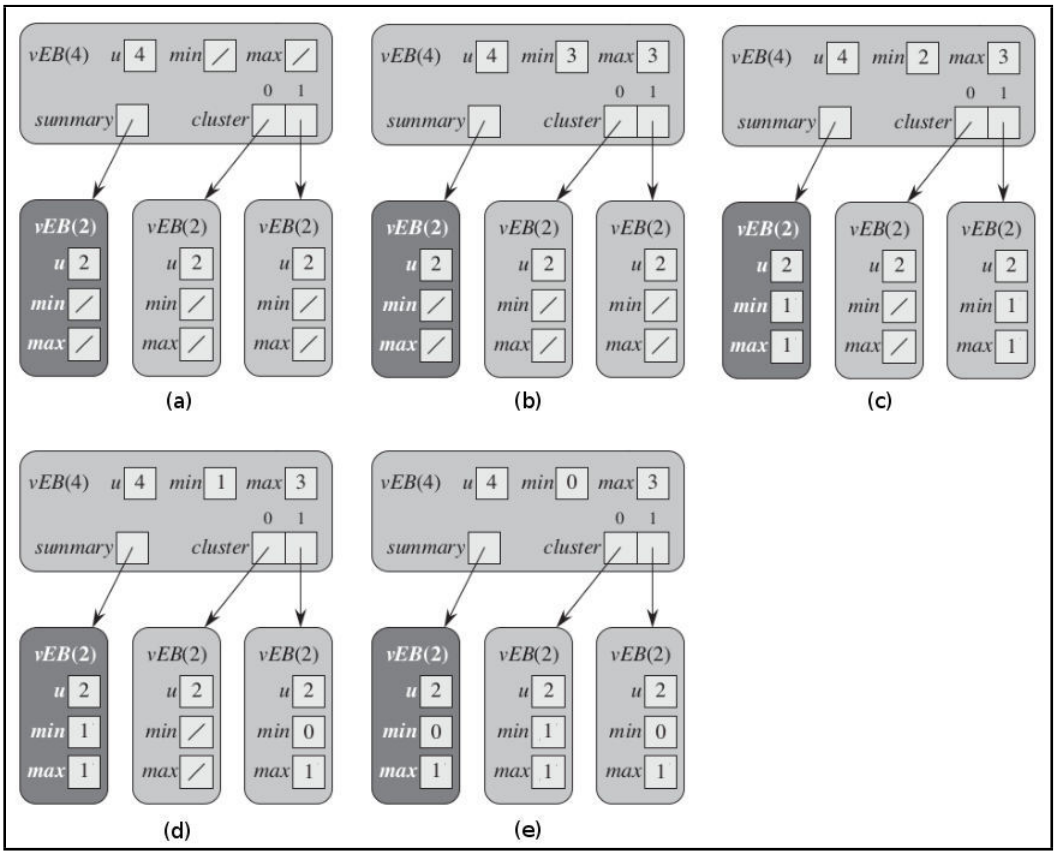

Figura 17 – Exemplo de inserção das chaves 3, 2, 1 e 0 na VEB
Fonte: Adaptado de Cormen *et al.* (2009)

```
1 veb_insercao_em_no_vazio(Q, x)
2     Q.min = x
3     Q.max = x
```

Quadro 11 – Exemplo de um pseudocódigo do procedimento de adição de um nó vazio em uma VEB
Fonte: Adaptado de Cormen *et al.* (2009)

campo *resumo* se faz necessária para contabilizar as subestruturas utilizadas. Sempre que um novo subconjunto é criado, seu índice é inserido no campo *resumo*, responsável por armazenar todos os índices dos subconjuntos não-vazios referenciados pelo vetor *cluster* a fim de otimizar a busca pelas chaves inseridas. Os itens (c), (d) e (e) da Figura 17 ilustram a configuração de uma árvore VEB(4) com a inserção de mais de uma chave.

A partir do segundo item inserido, uma nova chave sempre será direcionada para algum subconjunto de *cluster*. Caso a chave inserida expanda o limite inferior ou superior da estrutura, esta será posicionada no campo correspondente (*min* ou *max*) e enviará a antiga chave para o subconjunto apropriado em *cluster*. Os itens (d) e (e) ilustram a inserção das chaves 1 e 0 que expandem inferiormente o intervalo de valores da estrutura.

O pseudocódigo do procedimento de inserção de uma VEB pode ser visto no Quadro 12. O procedimento necessita basicamente da estrutura VEB alocada e do valor chave a ser inserido. Inicialmente é verificado se o nó em questão está vazio. Como a presença de pelo menos um item na estrutura implica no preenchimento dos limites inferior e superior dos valores do nó, a presença de um valor não nulo no campos *min* ou *max* indica a inserção de uma chave no nó. Caso o nó não possua chaves (linha 2) o procedimento auxiliar para a inserção de uma chave em um nó vazio é invocado, basicamente este procedimento atribui o valor chave aos campos *min* e *max*.

Caso a chave inserida seja menor que o limite inferior do nó, haverá uma troca entre as chaves inserida e a anteriormente posicionada no campo *min* (linha 5). A mesma situação ocorre para o limite superior(linha 13). Apesar do valor da maior chave já estar disponível no campo *max*, seu valor é replicado nas subárvores de *cluster*, caso o intervalo de valores da subárvore permita.

Em nós com capacidade superior a dois valores, uma chave é direcionada para os subconjuntos referenciados pelo *cluster* (linha 7), seja ela a nova chave ou a anteriormente posicionada em *min*. O subconjunto a receber a chave é calculado pelo procedimento high(), caso o subconjunto esteja vazio seu preenchimento será feito com a ajuda do procedimento veb_insercao_no_vazio (linha 10) e o índice do subconjunto será inserido no campo *resumo* (linha 9) para posteriormente facilitar uma busca. Caso o subconjunto seja não-vazio o procedimento de inserção continua recursivamente a partir do subconjunto calculado.

```
1  veb_insercao(Q, x)
2      se (Q.min = NULO)
3          veb_insercao_em_no_vazio(Q, x)
4      senao
5          se (x < Q.min)
6              troca(Q.min, x)
7          se (Q.u > 2)
8              se (veb_minimo(Q.cluster[high(x)]) = NULO)
9                  veb_insercao(Q.summary, high(x))
10                 veb_insercao_em_no_vazio(Q.cluster[high(x)],
       low(x))
11             senao
12                 veb_insercao(Q.cluster[high(x)], low(x))
13         se (x > Q.max)
14             Q.max = x
```

Quadro 12 – Exemplo de um pseudocódigo do procedimento de inserção de uma chave na VEB
Fonte: Adaptado de Cormen *et al.* (2009)

Como a cada iteração o procedimento restringe o intervalo de valores em $\sqrt{u}$, os casos base serão executados em no máximo $\log_2 \log_2 u$ iterações. Apesar da

chamada ao procedimento da linha 9 não usar como base o intervalo de chaves válidas inseridas na estrutura, mas sim aos índices preenchidos do *cluster*, o número de chamadas recursivas se equivale a linha 12 pois o subconjunto de atuação do procedimento possui o mesmo intervalo de valores.

A busca por uma chave na estrutura também é essencial para uma fila de prioridade durante o procedimento de atualização de chave. Inicialmente a busca em uma VEB se dá pela verificação dos campos *min* e *max* de cada nó, como visto no Quadro 13. A partir da estrutura e de um valor chave a ser pesquisado o procedimento de busca verifica se a chave requirida esta localizada nos campos delimitadores do nó (linha 2), em caso afirmativo um retorno verdadeiro é efetuado.

Em casos onde a chave não se encontra nos campos *min* ou *max* e o nó não possui subconjuntos para dar continuidade à pesquisa (linha 4), a árvore não possui a chave pesquisada. Caso o nó possua subconjuntos a serem pesquisados, o procedimento de busca será continuado a partir do subconjunto calculado pela função high(x) (linha 7). Como o intervalo do subconjunto é proporcional a $\sqrt{u}$, o valor da chave necessita ser ajustado para o novo intervalo, a função low(x) é a responsável por esta tarefa.

Como os casos base do procedimento são resolvidos em tempo constante e, a cada chamada recursiva o procedimento restringe a um custo constante o intervalo de domínio do nó em $\sqrt{u}$ do valor original u, o procedimento de busca por uma chave possui uma complexidade de tempo na ordem de $\log_2 \log_2 u$.

```
1 veb_membro(Q, x)
2     se (x = Q.min ou x = Q.max)
3         retorne verdadeiro
4     senao se (Q.u = 2)
5         retorne falso
6     senao
7         retorne veb_membro (Q.cluster[high(x)], low(x))
```

Quadro 13 – Exemplo de um pseudocódigo do procedimento de busca na VEB
Fonte: Adaptado de Cormen *et al.* (2009)

O processo de remoção de uma chave na VEB visa manter as mesmas premissas aplicadas à inserção quanto à configuração de um nó com uma, duas ou mais chaves. A remoção da única chave contida em um nó o deixa vazio, forçando a remoção de seu índice do *resumo* do seu relativo pai, caso ele exista. A transição do item (a) para o (b) da Figura 18 ilustra um exemplo deste caso com a remoção da chave com valor 1 (chave de valor 1 contida em *cluster*[0] de VEB(4)). Com a remoção da chave, o *cluster* passa a ficar vazio e a remoção de seu índice(0) em *resumo* se faz necessária.

A remoção de uma chave em um nó com apenas dois elementos implica no

ajuste do intervalo de valores do nó. Como apenas um elemento passa a estar contido no nó, esta chave remanescente definirá os limites inferior e superior do nó. Não é necessária a atualização do campo *resumo* do nó pai pois o nó envolvido na remoção ainda conterá um elemento. Como apenas uma chave passa a estar armazenada, os valores de *max* replicados em *cluster* devem ser apagados. Caso a chave de *max* seja o item removido, sua replicação também deve ser apagada. Caso a chave de *min* seja a removida, o valor de *max* passa a delimitar também o limite inferior e, a fim de respeitar a propriedade do campo *min*, sua chave não deve ser replicada no campo *cluster*. Devido a esta replicação de chaves não ser aplicada a nós VEB(2), neste caso a remoção dos itens em *cluster* não é necessária.

Remover a chave em um nó que possui mais de dois itens pode se enquadrar em dois casos: a remoção de uma chave com um valor intermediário entre as chaves dos campos *min* e *max* ou a remoção de um dos dois campos limite. A remoção de um item intermediário segue-se como uma busca pela chave. Como a chave a ser removida não se encontra diretamente no nó, a busca pela chave segue recursivamente pelo campo *cluster* até se obter um dos casos base descritos anteriormente. A remoção da chave 1 em VEB(4) ilustrada na transição do item (a) para o item (b) na Figura 18 exemplifica este processo.

Caso a chave a ser removida esteja em um dos campos *min* ou *max*, um novo valor deve ser direcionado para o campo. Com a remoção da chave do campo *min*, por exemplo, o menor valor contido nas subárvores de *cluster* deve ser o substituto da chave removida. A chave substituta basicamente é movida para o campo *min* e removida do seu nó original. Analogamente o procedimento de remoção da chave em *max* resulta no posicionamento da maior chave contida no campo *cluster*. Ilustrada na Figura 18, a remoção da chave 3 em VEB(4) na transição do item (b) para o item (c) exemplifica o processo de remoção de uma chave contida em *max*.

O procedimento de remoção de chaves na VEB recebe um nó da estrutura (inicialmente o nó raiz) e a chave a ser removida como parâmetros, assumindo que a chave a ser removida se encontra na estrutura. A função, como ilustrada no Quadro 14, inicia a verificação dos casos base com o tratamento da remoção da única chave de um nó, podendo ser vista nas linhas 2 a 4. Como o nó passa a não possuir chaves, os campos delimitadores do intervalo de valores do nó recebem o valor NULO.

Para nós que possuem um intervalo que abrange no máximo duas chaves, com a remoção de uma delas o intervalo passa a ser delimitado pela única chave restante, como visto nas linhas 5 a 10.

A remoção da chave contida no campo *min* faz necessário o preenchimento do campo com a próxima chave na escala crescente de valores. O *cluster* contendo o próximo item é localizado (linha 13), sua chave colocada no campo *min* (linha 15) e removida de seu antigo lugar no *cluster* (linha 16). Como a chave estava ajustada para

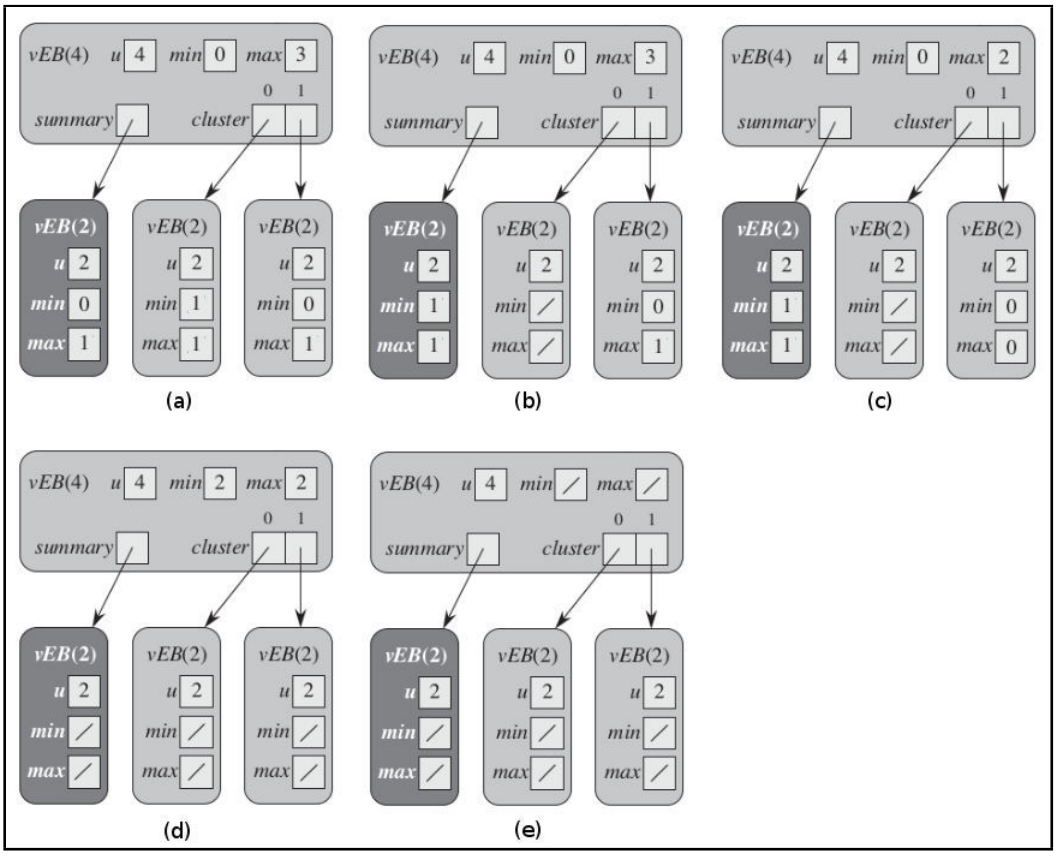

Figura 18 – Exemplo de remoção das chaves 1, 3, 0 e 2 em uma VEB, nesta ordem
Fonte: Adaptado de Cormen *et al.* (2009)

o intervalo suportado pelo nó em *cluster*, a função Index(x, y) ilustrada na Fórmula (3) calcula seu novo valor para o intervalo do nó atual.

A remoção de uma chave contida em um subconjunto referenciado pelo *cluster* (linha 16) pode resultar na remoção da última chave do referido nó. Caso isto ocorra (linha 17), o campo *resumo* deve ser atualizado e, caso a chave removida seja o maior valor do nó (linha 19), o campo *max* também deverá ser atualizado com uma nova chave, sendo ela a maior chave restante em *cluster* (linha 24) ou a chave contida em *min*, caso seja a única restante. Mesmo que após a remoção de um item contido no *cluster* ainda restem valores intermediários, caso a chave removida seja replicada do campo *max* este campo deve ser atualizado (linha 26).

O procedimento de remoção de uma chave em uma VEB possui uma complexidade de tempo na ordem de $O(\log_2 \log_2 u)$ visto que seus casos base são executados em $\Theta(1)$ e a cada chamada recursiva o procedimento reduz o intervalo de valores para $\sqrt{u}$ do valor u inicial.

Apesar de executar uma chamada recursiva na linha 16 e podendo executar outra na 18, uma delas sempre será executada em tempo constante. A execução da chamada recursiva da linha 18 depende do resultado ds condição da linha 17. Pode-se então definir duas situações: a condição da linha 17 é satisfeita ou não satisfeita.

Caso o resultado seja falso a remoção da chave não implica na atualização do campo *resumo* e a chamada da linha 18 não ocorre. Caso a condição da linha 17 seja verdadeira, a segunda chamada será executada em tempo $O(\log_2 \log_2 u)$ para atualizar o campo *resumo* porém, como o subconjunto da chave removida em *cluster* está vazio, a chamada ao procedimento da linha 16 executa o caso base das linhas 2 a 4, ou seja, uma chamada executada em tempo constante.

```
1  veb_remocao(Q, x)
2      se (Q.min = Q.max)
3          Q.min = NULO
4          Q.max = NULO
5      senao se (Q.u = 2)
6          se (x = 0)
7              Q.min = 1
8          senao
9              Q.min = 0
10         Q.max = Q.min
11     senao
12         se (x = Q.min)
13             primeiro_cluster = veb_minimo(Q.summary)
14             x = index(primeiro_cluster,
       veb_minimo(Q.cluster[primeiro_cluster]))
15             Q.min = x
16         veb_remocao(Q.cluster[high(x)], low(x))
17         se (veb_minimo(Q.cluster[high(x)]) = NULO)
18             veb_remocao(Q.summary, high(x))
19             se (x = Q.max)
20                 max_summary = veb_maximo(Q.summary)
21                 se (max_summary = NULO)
22                     Q.max = Q.min
23                 senao
24                     Q.max = index(max_summary,
       veb_maximo(Q.cluster[max_summary]))
25         senao se (x = Q.max)
26                 Q.max = index(index(high(x)),
       veb_maximo(Q.cluster[high(x)]))
```

Quadro 14 – Exemplo de um pseudocódigo do procedimento de remoção da VEB
Fonte: Adaptado de Cormen *et al.* (2009)

Apesar de não apresentar operações exclusivas de filas de prioridade, a combinação de algumas de suas operações, como vistas no Quadro 10, permitem os mesmos resultados. Como todas as operações envolvidas nas equivalências estão incluídas na mesma classe assintótica, as operações de filas de prioridade criadas mantêm-se na ordem de $O(\log_2 \log_2 u)$.

Apesar das equivalências entre as operações necessárias a uma fila de aprioridade e as operações oferecidas pela VEB, na teoria nem sempre se pode utilizar uma árvore VEB como fila de prioridade no Algoritmo de Dijkstra, visto que a VEB

exige que o valor de u seja constante, ou seja, o intervalo dos valores das chaves por ela armazenadas e por consequência o custo máximo que o caminho para os vértices pode assumir deve ser conhecido.

No contexto do Algoritmo de Dijkstra, o intervalo de valores para prioridade é variável, podendo assumir um valor $o(|V|.c)$ no pior caso, onde c é o custo máximo atribuído a um enlace pela métrica ETX e $|V|$ o total de vértices. A princípio não se sabe o número de vértices do grafo.

Na prática, porém, o protocolo para redes de malha sem fio utilizado neste trabalho, o OLSRD, define um limite para o custo no qual é viável utilizar uma rota. Mesmo que exista um caminho para um vértice, se este apresentar um custo superior ao definido pelo valor de saturação da métrica de roteamento adotada, esta rota será ignorada.

No protocolo OLSRD, o teto para o custo do caminho é limitado por uma constante de 4 bytes, ou seja, na prática, nenhum vértice terá prioridade maior que $2^{32} - 1$ (a saber, 4294967295) ainda que a rota até ele possua infinitos saltos.

Até o presente estado da arte, o protocolo OLSRD utiliza uma variação de uma árvore AVL como fila de prioridade para o Algoritmo de Dijkstra. Tal estrutura apresenta uma complexidade de $O(\log_2 |V|)$ para as operações de inserção, ExtraiMínimo e DecrementaChave.

Considerando a aplicação de uma VEB, no pior caso, a complexidade para estas mesmas operações seria $O(\log_2 \log_2 4294967295)$. A partir dessa constatação, pode-se concluir que teoricamente a VEB executaria as operações de fila de prioridade em no máximo 5 iterações, ou seja, em um tempo constante independente do tamanho do grafo.

3.7 ESTRUTURAS DE DADOS DO PROTOCOLO OLSRD

O processo de cálculo dos caminhos mínimos do OLSRD é dependente principalmente de 4 estruturas, sendo responsáveis por armazenar a topologia da rede e organizar logicamente os dados relacionados ao grafo e ao cálculo de caminhos mínimos propriamente dito.

As estruturas denominadas tc_edge_entry e tc_entry são responsáveis por armazenar as informações referentes a todos os roteadores e conexões da rede na forma de um grafo. Todo roteador pertencente à rede é representado por um nó do tipo tc_entry, responsável por armazenar todas as informações relevantes do vértice como seu ip (addr), custo (path_cost) e número de saltos (hops) do caminho a partir da origem (path_list_node), próximo salto no caminho (*next_hop) e um nó inicial para

as arestas (edge_tree), dentre outros.

As arestas do grafo são representadas por um tc_edge_entry que armazena as informações básicas de uma aresta, como endereço de destino (T_dest_addr) e custo da arestas (cost). Para fins de otimização, o OLSRD organiza o grafo usando uma *threaded* AVL, tanto para vértices quanto para arestas, garantindo a inserção, remoção e busca de um vértice ou aresta em um tempo na ordem de $O(\log_2(|A|))$.

A árvore *threaded* AVL do OLSRD é composta pelas estruturas avl_tree e avl_node. Uma avl_tree é responsável por indicar informações cruciais referentes à árvore, como seus nós raiz, primeiro (menor valor chave) e último (maior valor chave), além do número de nós da árvore. Cada nó (avl_node) da árvore possui alguns campos para organização da árvore, como nó pai, filho esquerdo e direito, nós anterior e próximo de acordo com o valor crescente das chaves, balanceamento, além do campo chave propriamente dito.

A organização dos vértices do grafo é feita com base no nó vertex_node do tipo AVL da estrutura tc_entry, sendo chaveada pelo seu endereço ip (addr). As arestas são organizadas pelo campo edge_node do tipo avl_node da estrutura tc_edge_entry, sendo chaveadas pelo seu endereço ip de destino (T_dest_addr). O nó raiz da árvore de arestas é salvo pelo campo edge_tree da estrutura tc_entry. A Figura 19 ilustra a organização das estruturas relacionadas à representação do grafo. As conexões ilustram a comunicação entre as estruturas e os campos envolvidos nesta relação.

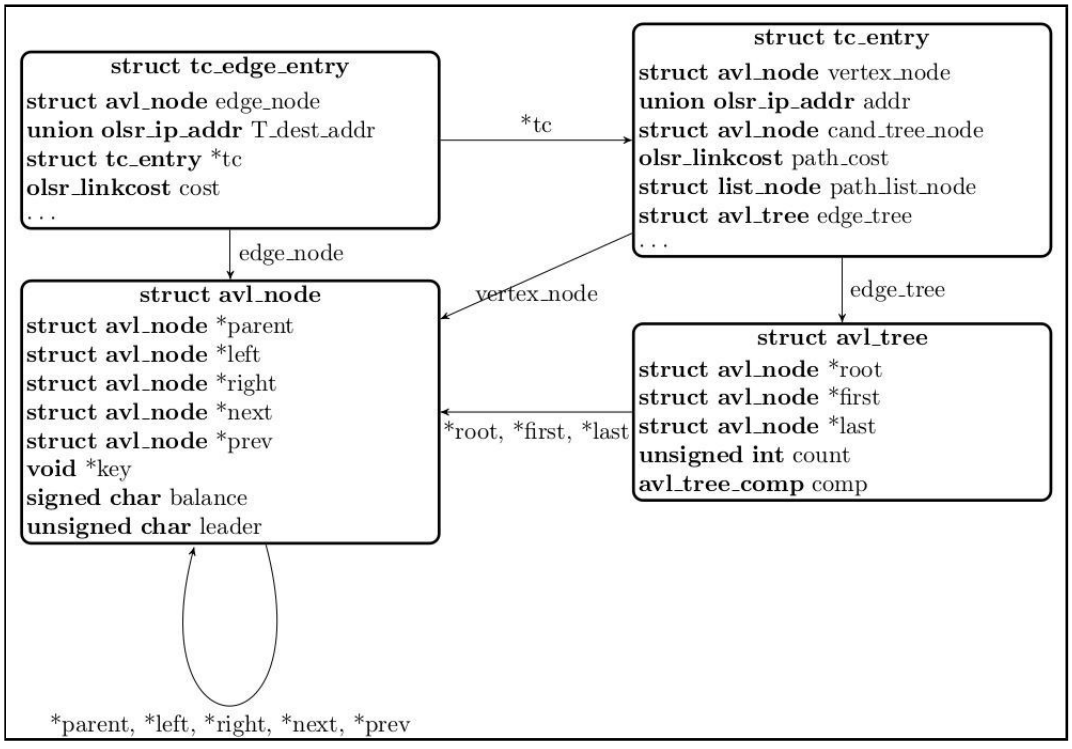

Figura 19 – Organização das estruturas de dados relacionadas à representação da topologia da rede no OLSRD.

Fonte: Autoria própria

Os campos edge_node da estrutura tc_edge_entry e vertex_node de tc_entry (ambos do tipo avl_node) foram estrategicamente posicionados como o primeiro campo da estrutura que os detêm, respectivamente. A partir de um nó avl_node pode-se ter acesso em tempo constante à estrutura na qual ele pertence (seja ela um vértice ou aresta) visto que o endereço do nó AVL é também o endereço da estrutura pai, bastando apenas alterar o tipo de dado que o endereço representa para transitar entre as estruturas.

Uma estrutura tc_entry também é responsável por armazenar nós de outras estruturas que serão utilizadas pelo algoritmo de cálculo de caminhos mínimos do OLSRD, tais como um nó da lista de final de caminhos (path_list_node) e um nó avl_node que servirá como um nó da fila de prioridade (cand_tree_node). A Figura 20 ilustra a organização das estruturas quanto ao processo de cálculo dos caminhos mínimos.

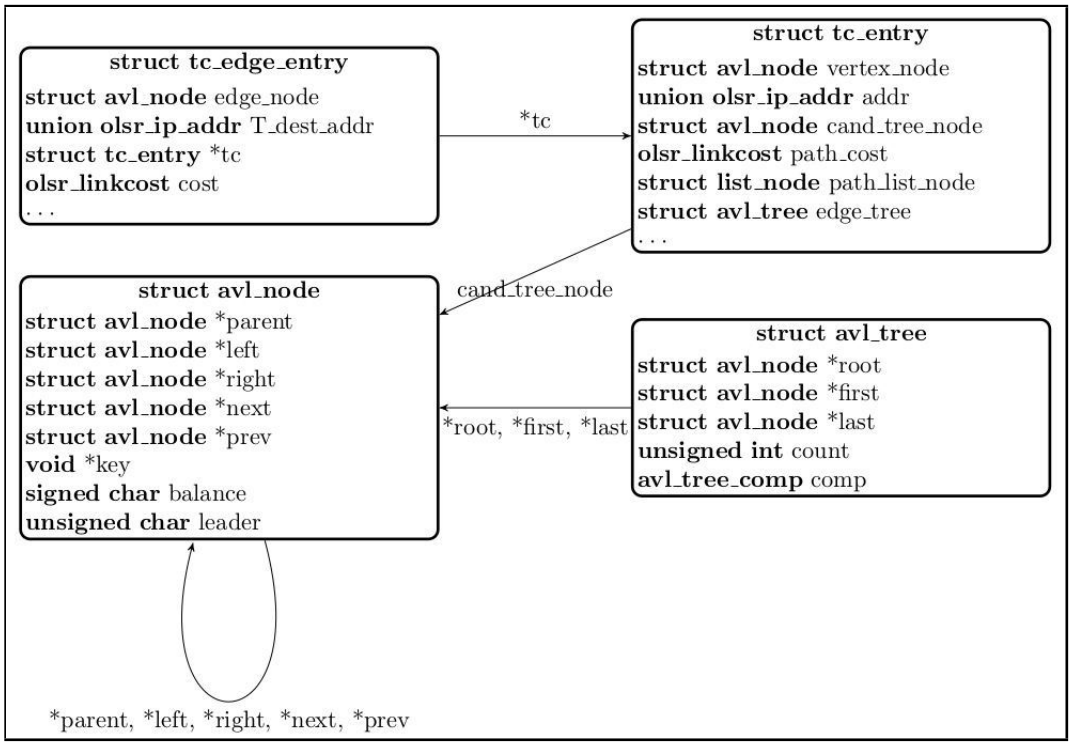

Figura 20 – Organização das estruturas de dados relacionadas ao cálculo de caminho mínimo do OLSRD.

Fonte: Autoria própria

O protocolo utiliza a mesma implementação da *threaded* AVL para organizar os vértices, arestas e os itens da fila de prioridade, variando apenas o tipo da chave utilizada (endereço IP para o grafo e a estimativa de custo do caminho mínimo para o vértice na fila de prioridade).

A escolha de uma *threaded* AVL em relação à uma árvore AVL simples se dá pela facilidade em percorrer todos os elementos da árvore iterativamente, procedimento útil ao processo de exploração de um vértice no Algoritmo de Dijkstra, quando

é preciso identificar todas as arestas do vértice a fim de tentar melhorar as estimativas de seus adjacentes.

4 DESENVOLVIMENTO

O desenvolvimento deste trabalho se dá primeiramente pela análise de complexidade do algoritmo de cálculo de caminhos mínimos do protocolo OLSRD, a fim de definir a classe assintótica das operações executadas pela árvore AVL, estrutura utilizada como fila de prioridade pelo Algoritmo de Dijkstra implementado no protocolo. Com base nessas classes assintóticas define-se um limite superior para o desempenho das filas de prioridades a serem estudadas.

Baseado nas filas de prioridade disponíveis na literatura e no desempenho apresentado pela AVL, decidiu-se estudar mais profundamente as estruturas heap binário e árvore Van Emde Boas. O heap apesar de teoricamente possuir a mesma ordem assintótica de uma árvore AVL, tradicionalmente possui constantes menores somadas à sua complexidade. Outro fator determinante para a escolha do heap binário foi devido a estrutura ser classicamente utilizada como fila de prioridade no Algoritmo de Dijkstra em diversos cenários.

A escolha pela Van Emde Boas deu-se pela competitiva complexidade apresentada graças à sua dependência de um fator não relacionado ao número de elementos armazenados na fila de prioridade, como ocorre com as duas outras estruturas, mas sim do intervalo de valores aceitos como chave nessa árvore.

Devido ao limite estabelecido pelo protocolo OLSRD para valores viáveis relacionados ao custo dos caminhos para os vértices, o custo máximo das chaves a serem inseridas na fila de prioridade passa a ser conhecido, contribuindo para a definição do intervalo [0, *u*-1] da Van Emde Boas. Como visto na Seção 3.6, dado o intervalo de chaves válidas a serem inseridas, a VEB apresenta um fator constante limitando sua complexidade no pior caso, resultando na teoria em um custo relativo a apenas $O(5)$ para executar as operações básicas da estrutura no protocolo OLSRD, independente da quantidade de itens armazenados nela.

Com base no estudo de Wollenberg (2012) que reporta a qualidade dos enlaces de uma rede *mesh*, foi possível representar as características uma rede de malha sem fio em um ambiente de testes. O Algoritmo de Dijkstra utilizando as filas de prioridades definidas foi submetido a uma análise de desempenho com base em grafos gerados a partir do padrão reportado pelo estudo.

Baseado nas estimativas de frequência e ciclos de processamento consumidos pelas operações de fila de prioridade executadas no Algoritmo de Dijkstra, obtidas na análise, juntamente com a classe assintótica das filas de prioridades estudadas, estimou-se o desempenho destas estruturas de dados no contexto de uma rede de malha sem fio.

Com base no desempenho e viabilidade de implementação das filas de pri-

oriodades no protocolo OLSRD, uma delas foi implementada no protocolo a fim de analisar o real desempenho da estrutura em uma rede de malha sem fio. A fim de se analisar o comportamento das filas de prioridades considerando também a mudança dinâmica no grafo de uma rede de malha sem fio, foram realizados testes com o protocolo OLSRD em ambientes emulados utilizando o emulador olsr_switch.

A Seção 4.1 apresenta os critérios definidos para o desenvolvimento do ambiente de teste utilizado neste trabalho, como escolha da estrutura de dados utilizada na representação do grafo e detalhes de implementação aplicados no ambiente de teste a fim de manter as características do Algoritmo de Dijkstra compativeis com a implementação encontrada no protocolo OLSRD.

A Seção 4.2 descreve a organização das estruturas de dados utilizadas pelo Algoritmo de Dijkstra com a árvore AVL como fila de prioriodade. A seção apresenta as características particulares da AVL utilizada pelo OLSRD, detalhes de implementação e análise das complexidades inerentes às operações executadas pela estrutura. Assim como na Seção 4.2, as Seções 4.3 e 4.4 descrevem as características relacionadas às estruturas heap binário e árvore Van Emde Boas respectivamente.

A Seção 4.5 descreve as características da rede de malha sem fio utilizada como base para a modelagem do cenário utilizado para a análise de desempenho das estruturas. A partir de distribuições de probabilidade baseadas em dados de uma rede de malha sem fio, coletados por Wollenberg (2012), redes maiores foram geradas para avaliar o desempenho das estrutura de dados utilizadas.

4.1 AMBIENTE DE TESTE

A partir da seleção dos algoritmos, o próximo passo foi a verificação da viabilidade dos algoritmos no contexto de uma rede de malha sem fio. Para tal, foi desenvolvido um ambiente de teste modularizado a fim de facilitar a implementação, teste e análise de desempenho.

O ambiente desenvolvido foi projetado mantendo as características e a organização das estruturas de dados envolvidas no processo do cálculo de caminho mínimo apresentadas pelo OLSRD, como visto na Figura 20. Para fins de simplificação além de impedir a interferência de fatores externos ao Algoritmo de Dijkstra, foram removidos todos os componentes desnecessários ao algoritmo, como campos relacionados ao processo de descoberta e gerenciamento de conexões.

Optou-se também por substituir a representação original do grafo feita por uma árvore AVL para uma clássica lista de adjacência, visto sua simplicidade além de não interferir no desempenho das filas de prioridade visto sua implementação original.

Assim como na implementação original, buscou-se incorporar o item da fila de prioridade na estrutura responsável pela representação do vértice. Apesar de originalmente o Algoritmo de Dijkstra ser a melhor solução para o problema do menor caminho de origem única, uma rede de malha sem fio é representada pela variação dinâmica deste problema (DSSSP - Dynamic Single Source Shortest Paths).

Devido a constante execução do Algoritmo de Dijkstra, a cada chamada uma nova fila de prioridade teria de ser alocada e ao fim do algoritmo ser devolvida a memória ao sistema, demandando um tempo extra e desnecessário à esta operação. Outra vantagem é o acesso em tempo $\Theta(1)$ ao nó relacionado a vértice na fila de prioridade. A inclusão dos nós da fila de prioridade na estrutura dos vértice dispensa a constante alocação dos mesmos, deixando para a fila de prioridade apenas a organização lógica dos elementos.

4.2 IMPLEMENTAÇÃO DA ÁRVORE AVL

A partir do ambiente de teste desenvolvido, iniciou-se o processo de implementação das filas de prioridade a fim de analisar seus desempenhos. Para garantir uma representação fidedigna da árvore AVL utilizada no protocolo utilizou-se a mesma biblioteca responsável pela estrutura implementada no OLSRD. Assim como no protocolo, os nós da árvore AVL foram incluídos estaticamente na estrutura responsável pela representação dos vértices, isentando os procedimentos da AVL de gerenciar a alocação dos nós para a árvore.

Com base em um cenário projetado para respeitar as características apresentadas no protocolo OLSRD, poucas alterações foram necessárias para a adaptação da árvore AVL como fila de prioridade do Algoritmo de Dijkstra projetado. A Figura 21 ilustra a organização das estruturas de dados do Algoritmo de Dijkstra utilizando a árvore AVL como fila de prioridade.

As estruturas *vertice* e *aresta* são utilizadas para a representação do grafo gerado pela topologia da rede. Os vértices ilustram os roteadores ativos na rede e as arestas representam as conexões entre eles. A estrutura desenvolvida para o grafo do ambiente experimental abstrai apenas os campos relevantes para a representação da topologia e para o cálculo dos caminhos mínimos. Como a representação da rede é requisitada globalmente no protocolo, outros atributos são incluídos nas estruturas citadas de acordo com a necessidade de outros módulos do protocolo. Atributos estes, desnecessários e irrelevantes ao prosseguimento do estudo, sendo então, omitidos no ambiente de teste.

A estrutura *vertice* possui um campo do tipo avl_node, utilizado pela fila de prioridade. Sua inclusão do campo dentro da estrutura *vertice* desobriga os procedi-

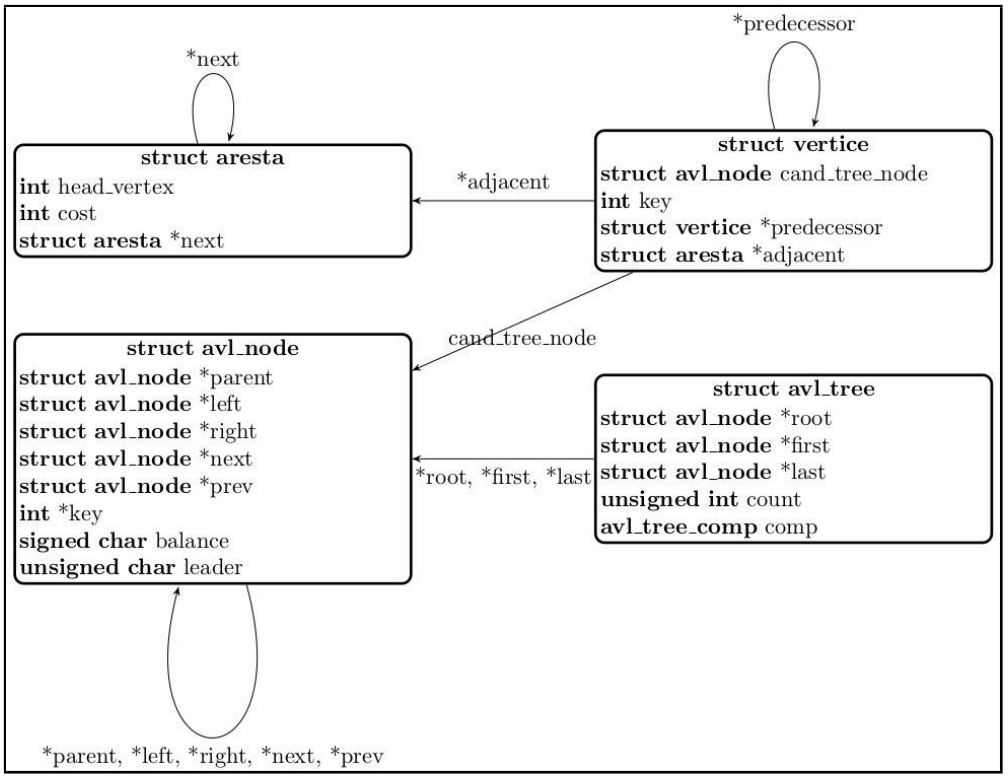

Figura 21 – Organização do Dijkstra experimental utilizando uma árvore AVL.
Fonte: Autoria própria

mentos da AVL de gerenciar a alocação e liberação de memória para estes nós visto que os nós da fila de prioridade são alocados juntamente com os vértices. A estrutura possui também um campo *key* utilizado para a identificação do vértice. No ambiente de teste este identificador é um número pertencente ao conjunto dos números naturais, no protocolo porém, representa o endereço IP do roteador.

O campo *predecessor* representa o atributo π referente ao vértice antecessor no caminho mínimo que o Algoritmo de Dijkstra atribui a cada vértice, como descrito com mais detalhes na Seção 3.3. O atributo d também atribuído pelo Algoritmo de Dijkstra a cada vértice está implícito no campo *key* de *cand_tree_node*. O campo *adjacent* inicia a lista de arestas de saída do vértice.

A estrutura *aresta* descreve as conexões entre os vértices. O campo *head_vertex* possui o identificador do vértice no qual a aresta incide, o campo *cost* descreve a qualidade da conexão e o atributo *next* conecta o nó à próxima aresta incidente no vértice.

A estrutura *avl_tree* é a responsável por contabilizar informações importantes relacionadas à árvore AVL. A estrutura por meio dos campos *root*, *first* e *last* identifica respectivamente o nó raiz da árvore, o elemento mais à esquerda e mais à direita da árvore. O campo *count* contabiliza o número de itens presentes na AVL.

A *avl_node* é a estrutura responsável por representar os nós do tipo AVL. Os campos *parent*, *left* e *right* representam respectivamente a relação do nó com seus relativos nó pai, filho esquerdo e filho direito. Os campos *next* e *prev* são utilizados

pelos procedimentos relacionados à propriedades da *threaded* AVL, indicando os nós próximo e anterior relativos ao nó em questão. O atributo *key* representa a chave utilizada pela árvore que no Algoritmo de Dijkstra representa a prioridade dos vértices. O campo *balance* contabiliza o balanceamento do nó e o atributo *leader* indica o estado do nó quanto a uma lista de outros nós com a mesma prioridade: 1 (um) significa que o nó em questão é único ou é o primeiro da lista; 0 (zero) representa um nó com chave duplicada na cauda da lista.

Árvores AVL foram projetadas inicialmente para armazenar valores chave distintos, característica esta que não pode ser garantida quando os valores inseridos na estrutura são relacionados aos custos do vértices durante a execução do Algoritmo de Dijkstra.

O critério de inserção de chaves repetidas em um dos lados da árvore como em uma BST pode levar a AVL, após algumas rotações, a uma configuração que não respeita a propriedade de uma árvore binária de pesquisa. A Figura 22 ilustra a inserção de uma sequência de chaves que leva a árvore a tal configuração. O critério adotado na figura é a manutenção de itens repetidos sempre à direita e, após uma rotação este critério é violado.

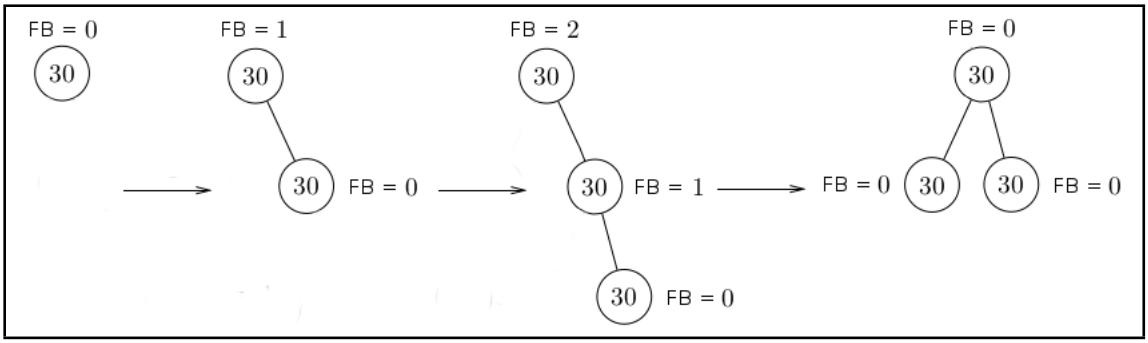

Figura 22 – Representação da inserção de uma sequência de chaves repetidas que levam a AVL a um estado de inconsistência.
Fonte: Autoria própria

Uma solução clássica para controle de chaves repetidas em um AVL é a adição de um campo contador à estrutura de cada nó para contabilizar o número de chaves contidas nele, evitando a alocação de um novo nó para chave repetida. Esta abordagem porém, não pode ser aplicada em um fila de prioridade do Algoritmo de Dijkstra, pois cada chave está associada a um vértice do grafo, ou seja, deve ser tratada individualmente.

Para contornar esta situação a implementação da AVL disponibilizada pelo protocolo organiza as chaves duplicadas em listas conectadas ao primeiro nó AVL inserido com a referida chave. A partir da adição desta lista, os filhos esquerdo e direito passam a referenciar apenas chaves distintas do nó pai. A Figura 23 ilustra a organização de uma AVL a partir da inserção de itens com as chaves 55 e 60 já

existentes na árvore.

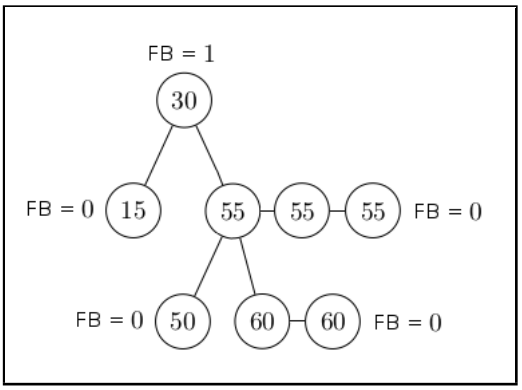

Figura 23 – Representação de uma AVL com a repetição dos itens com chaves 55 e 60. Fonte: Autoria própria

Os procedimentos de inserção(avl_insert) e remoção(avl_delete) da árvore AVL são descritos no Apêndice A. Considerando as operações de comparação efetuadas pelo procedimento *avl_insert* somadas às operações realizadas pelos procedimentos auxiliares, o procedimento possui uma complexidade esperada de $O(7 \cdot \log_2 n)$. Analisando o pior caso do algoritmo porém, pode-se obter operações com complexidade $O(n)$ relacionadas à busca pela posição do nó na lista de itens de mesma prioridade nas linhas 31 a 33, apesar de ser possível adotar outra estratégia de inserção em tempo $\Theta(1)$.O procedimento de remoção, somado à complexidade dos procedimentos auxiliares apresenta uma complexidade esperada de $O(12 \cdot \log_2 n)$.

Apesar da árvore AVL não ter sido originalmente projetada como uma fila de prioridade, não tendo um procedimento de atualização de um valor chave já armazenado na estrutura, pode-se facilmente contornar esta situação. O procedimento DecrementaChave, responsável por tal operação, implementado na AVL do protocolo utiliza-se da combinação dos procedimentos de remoção e inserção de nós na árvore, resultando em um complexidade esperada de $O(19 \cdot \log_2 n)$, ou no pior caso $O(n)$. Dado um nó que seja submetido ao procedimento de atualização de chave, este é removido da árvore e reinserido com o valor chave atualizado. A complexidade do procedimento DecrementaChave é a soma da complexidade dos procedimentos de inserção e remoção.

Devido à peculiaridade da árvore AVL utilizada ser implementada como uma *threaded* AVL (visto na Subseção 3.5.2), isto permitiu o acesso ao elemento com menor valor chave da árvore em tempo $\Theta(1)$ a partir do campo *first* da estrutura avl_tree, apesar de adicionar um custo computacional (mesmo que constante) aos procedimentos de inserção e remoção da árvore referente à atualização dos campos *prev* e *next* dos nós envolvidos no procedimento. Esta característica permitiu ao procedimento ExtraiMínimo uma complexidade de $O(12 \cdot \log_2 n)$ no pior caso.

4.3 IMPLEMENTAÇÃO DO HEAP BINÁRIO

Apesar do heap binário ser tradicionalmente implementado em um espaço contíguo de memória, como um vetor (Figura 6), esta abordagem se tornaria inviável em uma rede *mesh* devido à necessidade de se conhecer previamente o tamanho do espaço de memória a ser alocado para a estrutura; como a quantidade de elementos na fila de prioridade varia de acordo com o número de roteadores ativos na rede, se torna impossível a definição de um limite superior para a capacidade da fila de prioridade.

A partir destas constatações optou-se por uma abordagem dinâmica para a representação do heap baseada em uma árvore binária quase completa (Figura 5). A implantação do heap deu-se basicamente pela implementação das estruturas de árvore binária, sendo necessário apenas a modificação dos procedimentos de inserção para manter a propriedade de uma árvore quase completa além da organização dos nós de acordo com a sua prioridade.

Baseada nestas modificações, a Figura 24 apresenta as estruturas necessárias ao heap binário, além de sua relação com as estruturas referentes ao grafo.

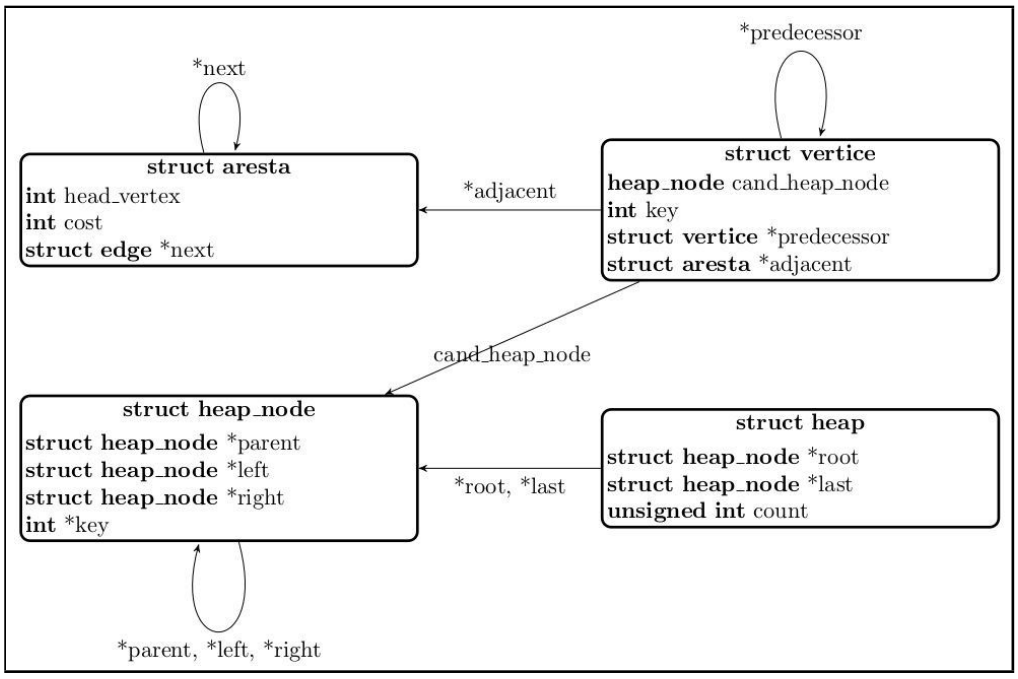

Figura 24 – Organização do Dijkstra experimental utilizando um heap binário.
Fonte: Autoria própria

A estrutura *heap* é a responsável pelo gerenciamento do heap binário, contabilizando o número de itens da fila e controlando os nós posicionados na raiz e na última posição do heap.

A estrutura *heap_node* representa um nó do heap, sendo este alocado es-

taticamente em cada vértice do grafo como a variável cand_heap_node na estrutura *vertice*. O campo key em *heap_node* representa a prioridade do vértice associado e o campo key na estrutura *vertice* representa o identificador do rodeador. Os campos *parent*, *left* e *right* simulam o resultado das funções pai(), esquerdo() e direito() respectivamente, utilizadas por um heap binário estático como descritas na Seção 3.4. A relação entre as estruturas *vertice* e *aresta* segue-se como descrita na Seção 4.2.

Pelo fato do heap não iniciar um novo nível em sua árvore antes de completar o último nível, ele garante uma profundidade igual à $\log_2 n + 1$. Com base neste teto para a altura da estrutura, a partir do número de nós incluídos no heap consegue-se identificar quantos elementos estão no último nível da árvore, podendo assim encontrar a posição na qual um novo elemento será incluído ou qual nó substituirá um nó removido, auxiliando os procedimentos de inserção e remoção respectivamente.

O Apêndice B apresenta os procedimentos relacionandos ao heap binário desenvolvidos neste trabalho. Contabilizando as chamadas para procedimentos auxiliares, o processo de remoção do item de maior prioridade na fila é executado em um tempo relativo a $O(13 \cdot \log_2(n))$ baseado no número de comparações efetuadas.

O procedimento de atualização de chave (heapDecreaseKey) apresenta uma laço iterativo para a troca de posição entre os nós pais e filho a um custo de 8 comparações a cada iteração do procedimento. Devido a um heap com n itens possuir $\log_2 n$ níveis, resultando em no máximo $log_2 n$ iterações, o procedimento heapDecreaseKey possui uma complexidade de $O(log_2 n)$

A partir da análise das operações assintoticamente relevantes executadas pelo procedimento de inserção heapInsert (linhas 11 e 21), o procedimento apresenta uma complexidade de tempo na ordem de $O(10 \log_2 n)$, oriunda das complexidades apresentadas por heapFindParentInsertNode e heapDecreaseKey.

4.4 ADAPTAÇÃO DA VAN EMDE BOAS COMO FILA DE PRIORIDADE

Apesar da Van Emde Boas possuir uma equivalência com uma fila de prioridade (como visto no Quadro 10), algumas alterações são necessárias para adequar a estrutura à uma fila de prioridade útil ao Algoritmo de Dijkstra. A partir de algumas situações inerentes ao modelo adotado encontradas no processo de implementação, descritos a seguir, foram propostas algumas soluções descritas a seguir.

4.4.1 O Problema das Chaves Duplicadas

Originalmente a árvore VEB assume que os valores das chaves inseridas na estrutura são distintos, ou seja, não há repetição de chaves. Tal limitação impossibilita a utilização da árvore no Algoritmo de Dijkstra visto que não se pode garantir esta propriedade no cenário de uma rede de malha sem fio. Como as chaves da fila de prioridade são as estimativas de custo mínimo dos vértices durante a execução do algoritmo, é possível que vértices diferentes possuam valores iguais para suas estimativas.

Apesar deste problema ser facilmente corrigido adicionando dois novos campos responsáveis por contabilizar o número de chaves com o mesmo valor de cada nó (um contador para chaves com valor igual à *min* e outro para *max*), como proposto por Cormen *et al.* (2009), permitindo assim que a VEB aceitasse chaves duplicadas, ainda não seria possível utilizá-la no Algoritmo de Dijkstra.

A fila de prioridade do Algoritmo de Dijkstra não é responsável somente por informar qual a menor estimativa de custo dentre os vértices visitados, mas principalmente qual o vértice associado a esta, até então, estimativa de custo. A identificação do vértice é essencial para execução do algoritmo, sendo ele o vértice a ser explorado na próxima iteração.

A partir de tal constatação não se pode gerenciar apenas os valores das chaves da fila de prioridade, mas também quais vértices estão associados a elas. Para contornar este problema, todo campo chave de um nó VEB (*min* e *max*) deve conter uma lista responsável por armazenar e referenciar todos os vértices que possuem o mesmo valor chave, ou seja, ao adicionar uma nova chave também é adicionado ao nó VEB, portador da chave, um nó de uma lista encadeada responsável por referenciar o vértice dono da estimativa. A Figura 25 ilustra a estrutura de um nó VEB adaptado para o Algoritmo de Dijkstra.

Apesar de teoricamente uma lista encadeada garantir uma inserção em tempo constante, a busca e, por consequência, a remoção de algum de seus itens têm complexidade de tempo de $O(n)$, sendo n o número de itens. Adicionar uma subestrutura com complexidade linear à VEB prejudica a complexidade $O(\log_2 \log_2 u)$ tradicionalmente obtida, transformando a VEB em uma estrutura de complexidade linear no pior caso.

Assim como a implementação da AVL e do heap binário no Algoritmo de Dijkstra experimental, os nós do tipo lista estarão alocados dentro da estrutura do vértice, sendo apenas organizados logicamente durante a execução do algoritmo. Esta inclusão permite um acesso em tempo constante ao nó, não necessitando uma busca linear na lista. Desta forma, o procedimento de remoção de um elemento na lista se dará em

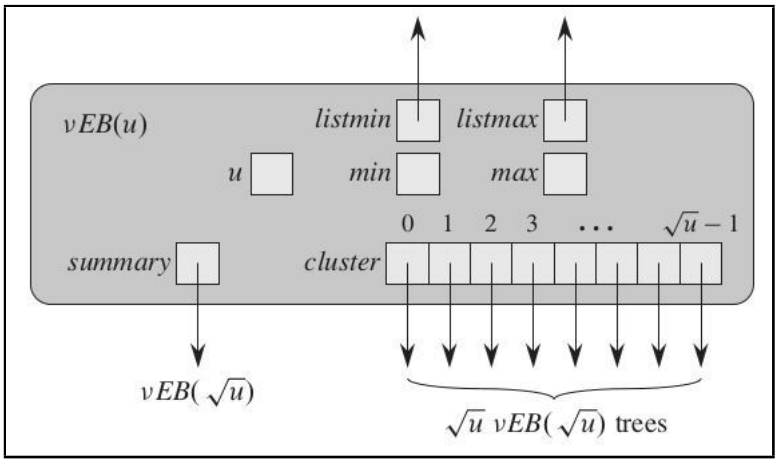

Figura 25 – Estrutura de um nó VEB adaptada para o Algoritmo de Dijkstra.
Fonte: Adaptado de Cormen *et al.* (2009)

tempo constante, devolvendo à VEB a complexidade na ordem de $O(\log_2 \log_2 u)$.

Outra característica apresentada pela VEB proposta por Cormen *et al.* (2009) é a duplicação do valor do campo *max* de um nó por toda a sua subestrutura de filhos. Como visto na Seção 3.6, diferentemente do valor de *min* armazenado apenas no nó mais próximo à raiz, o valor de *max* é duplicado nos nós pertences ao *cluster* sem nenhuma justificativa.

Não duplicar a chave *max* não altera a ordem de complexidade de tempo do algoritmo, porém possibilita a diminuição do número de nós alocados.

4.4.2 O Problema da Alocação Dinâmica de Nós VEB

A adaptação da VEB para permitir a duplicidade de chaves a habilita a ser utilizada como fila de prioridade do Algoritmo de Dijkstra, porém outro problema inviabiliza sua utilização em um protocolo de rede de malha sem fio.

Na árvore Van Emde Boas apresentada por Cormen *et al.* (2009), todos os nós da árvore necessários para armazenar as u chaves da estrutura são previamente alocados, mesmo que nenhuma chave no intervalo $[min, max]$ seja armazenada no nó. Esta configuração atribui para os procedimentos de inserção e remoção apenas a organização das chaves, isentando esses procedimentos do gerenciamento de ponteiros e da alocação e liberação de memória utilizada pelos nós.

A prévia alocação de todos os nós inviabiliza a implementação da estrutura no protocolo OLSRD, considerando-se a limitação de memória geralmente apresentada por um roteador. A única alternativa para viabilizar a utilização da VEB é implementar uma alocação de nós sob demanda.

Diferentemente das demais filas de prioridade apresentadas neste trabalho,

em que cada nó da estrutura (tanto um nó de um heap binário quanto um nó AVL) ocupa um espaço fixo na memória, permitindo que seja alocado juntamente com um vértice, um nó VEB não dispõe de tal atributo.

Apesar de um nó VEB armazenar dois valores chave, e por consequência duas listas de vértices detentores de tais valores, esta peculiaridade não altera o espaço de memória ocupada pelo nó. Entretanto, como cada nó é responsável por armazenar as chaves de um determinado subintervalo da estrutura, o número de nós filhos necessárias para isto dependerá do intervalo u de cada nó, impactando diretamente no número de referências que o vetor *cluster* deverá possuir, logo, alterando o tamanho do nó.

Como o tamanho do campo *cluster* é variável dependendo da sua localização na árvore (quanto mais próximo da raiz, maior), não há como seguir a mesma metodologia da AVL e do heap binário quanto à alocação dos nós junto ao vértice.

A Figura 26 apresenta um exemplo de uma árvore VEB adaptada com u igual 4 armazenando as chaves 1, 2 e 3, respeitando a nova organização da árvore de acordo com as mudanças propostas. A nova árvore não apresenta a alocação de nós desnecessários nem a duplicação das chaves de maior valor de cada nó. Pode-se notar a não existência de um nó VEB associado à primeira posição do vetor *cluster* no nó raiz além da não duplicação da chave 3 no nó associado à segunda posição do vetor *cluster*.

Os nós também apresentam exemplos da utilização da lista de vértices, utilizada para associar os vértices com suas chaves. Pode-se notar que a chave com valor 1 no nó raiz está associada aos vértices com os identificadores 5 e 1, e os vértices 2 e 3 à chave de custo 2, localizada em no nó VEB(2) com seu valor atualizado para o intervalo de valores suportados pelo nó. Os mesmos critérios apresentados são utilizados nos nós VEB associados ao campo resumo(*summary*), porém como não há vértices associados a estes nós uma chave genérica é inserida na lista de vértices destes nós, como o valor 1 definido como padrão.

A partir de tais soluções pode-se aplicar a Van Emde Boas no ambiente de teste do Algoritmo de Dijkstra. A Figura 27 ilustra o relacionamento entre as estruturas que compõe o Dijkstra experimental e as estruturas relacionadas à VEB.

O relacionamento das estruturas *vertice* e *aresta* segue como descrito na Seção 4.2. Diferentemente das estruturas anteriores, um nó VEB não pôde ser alocado estaticamente dentro da estrutura *vertice*. Como o nó VEB possui tamanho variável devido ao campo *clusters*, um vetor de ponteiros para as subestruturas filhas do nó, sendo dependente do intervalo u indicado, apenas o nó lista pode ser incluído na estrutura do vértice. O campo *key* da lista referencia o vértice detentor do nó e os campos *listMin* e *listMax* da VEB iniciam a lista de nós com estimativas de custo referentes à *min* e *max* respectivamente.

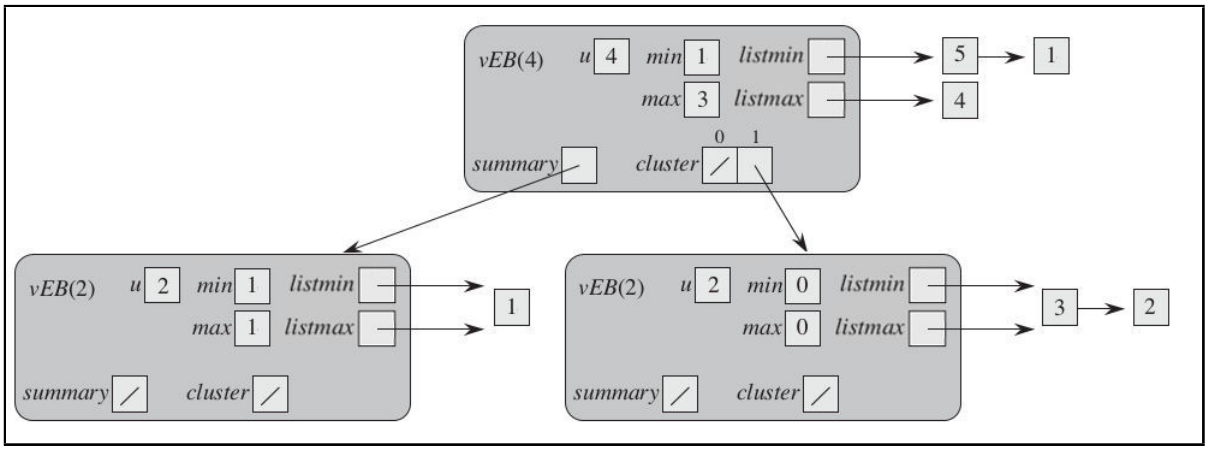

Figura 26 – Exemplo de uma árvore Van Emde Boas adaptada com u igual a 4 armazenando as chaves 1, 2 e 3.
Fonte: Adaptado de Cormen *et al.* (2009)

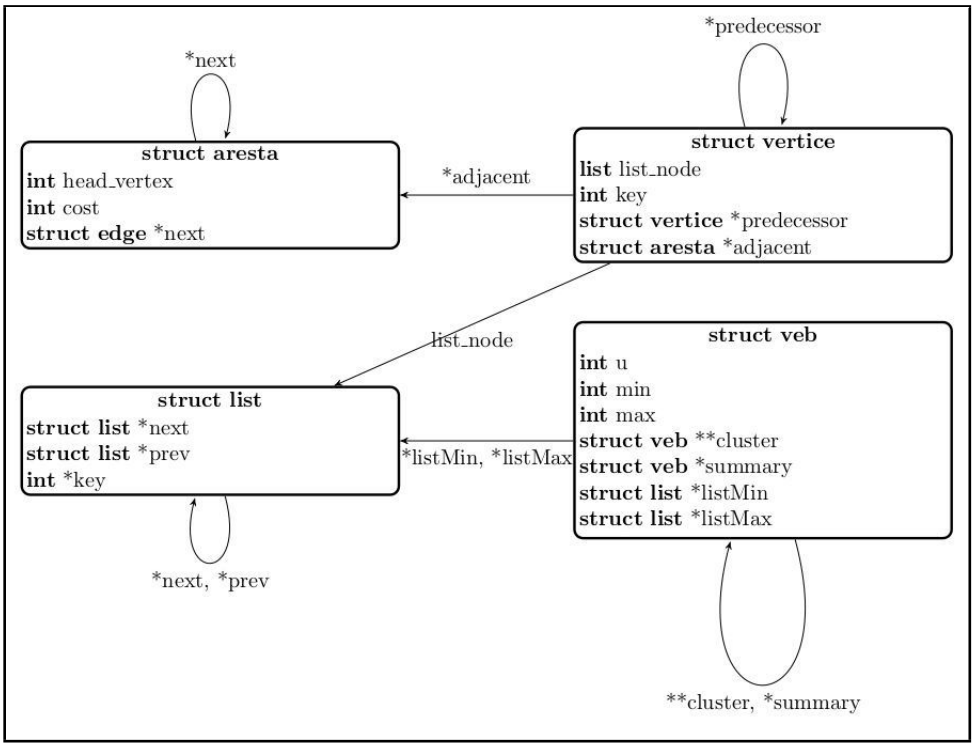

Figura 27 – Organização do Dijkstra experimental utilizando uma VEB.
Fonte: Autoria própria

Devido ao uso de uma lista para identificar os vértices detentores da chave usada na fila de prioridade, foi adicionado um espaço relativo a $O(n)$ à estrutura. Considerando o espaço $O(\log_2 \log_2 u)$ utilizado pela VEB para sua organização, a Van Emde Boas apresenta uma complexidade de espaço igual a $O(n) + O(\log_2 \log_2 u)$. Como o valor de u é constante, considera-se a complexidade de espaço na ordem de $O(n)$, a mesma das demais estruturas estudadas.

Diferentemente das demais estruturas apresentadas, os procedimentos de inserção e remoção na VEB necessitam gerenciar a alocação e liberação de memória

dos nós utilizados. As linhas 2 a 14 do procedimento de inserção na VEB, apresentado no Apêndice C, ilustram o tratamento para a alocação de um novo nó. A linha 3 executa a alocação do nó e as linhas 8 a 12 a alocação do vetor *cluster*, se necessário.

Com base nos dados apresentados no Apêndice C, a complexidade da operação de inserção em uma Van Emde Boas é $O(8 \log_2 \log_2 u)$. A operação de remoção de uma chave apresenta uma complexidade de $O(7 \log_2 \log_2 u)$ considerando os mesmos critérios. Devido à falta de operações específicas de filas de prioridade, a combinação destas operações foi necessária para a avaliação das operações DecrementaChave e ExtraiMínimo, conforme as equivalências apresentadas no Quadro 10.

A operação ExtraiMínimo apresentou uma complexidade de $O(7 \log_2 \log_2 u)$ e a operação DecrementaChave, resultante da combinação das operações de remoção e inserção, apresentou uma complexidade de $O(15 \log_2 \log_2 u)$.

Para adaptar a implementação da VEB ao ambiente de teste foi necessária a redução do número de bits de u de 32 para 16. Apesar do número de bits ter sido reduzido pela metade, apenas o último nível da árvore foi descartado, reduzindo o número de níveis da árvore de 5 para 4, resultando na prática na redução de apenas uma iteração se comparada com o comportamento que a estrutura teria no protocolo.

4.5 CONJUNTO DE BASE PARA TESTE

Para analisar o desempenho das filas de prioridade considerando as características de uma rede *mesh* foi construída uma configuração de teste baseada nos dados da rede de malha sem fio colaborativa da Opennet (2015), situada na cidade de Rostock na Alemanha. Estes dados foram coletados e divulgados por Wollenberg (2012).

Os dados foram coletados no intervalo de 1/1/2011 à 31/05/2011, totalizando um período de 3664 horas. A rede neste período possuía cerca de 300 roteadores cadastrados, todos executando o protocolo OLSR. A cada hora um registro contendo a configuração da rede foi armazenado, esta configuração indicava os roteadores ativos na rede, além das conexões entre eles. A qualidade das conexões foi registrada com base na métrica *link quality*. A métrica *link quality* é descrita por meio de dois valores, *Neighbor Link Quality*(NLQ) e *Link Quality*(LQ), que variam de 0 a 1 equivalentes a 0 a 100%. NLQ indica a probabilidade de que um pacote enviado para um roteador adjacente seja entregue com sucesso e LQ indica a probabilidade de sucesso no caminho inverso.

Para este trabalho porém, foi utilizada a métrica ETX, assim como no protocolo OLSRD, para estimar a qualidade das conexões. A métrica ETX pode ser obtida com

base nos dados da métrica *link quality* de acordo com a fórmula $1/(NLQ \cdot LQ)$.

A rede apresentou um total de 259 roteadores ativos durante o período de coleta dos dados, possuindo em média 175 roteadores ativos e 848 conexões a cada coleta dos dados. A Figura 28 mostra uma representação da rede em um dia arbitrário. O exemplo ilustrado apresenta 182 roteadores e 946 conexões detectadas pelo protocolo OLSR de um dos roteadores. A coloração das conexões representa sua qualidade de acordo com o *link quality*, cores mais escuras representam uma melhor conexão.

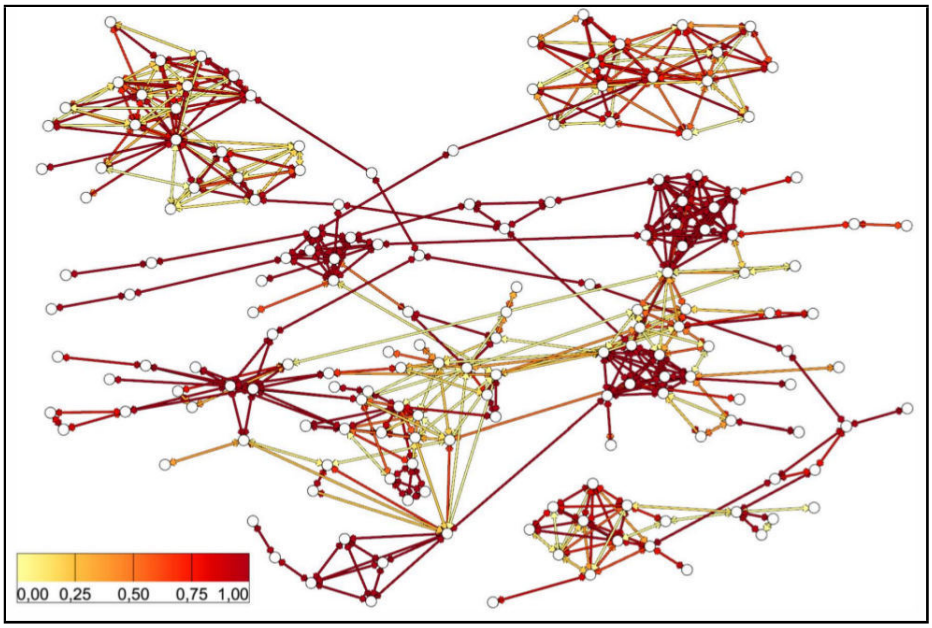

Figura 28 – Organização dos roteadores.
Fonte: Wollenberg (2012)

Com base nos valores reportados pode-se identificar características de uma rede *mesh* real, tais como grau médio de conexões de cada roteador e qualidade de cada conexão. O Gráfico 1 apresenta um histograma com o número médio de conexões por roteador. Pode-se notar o baixo grau de conectividade dos roteadores, já que cerca de 50% deles possuem no máximo 2 conexões. O Gráfico 2, por sua vez, apresenta a qualidade destas conexões, podendo-se observar que cerca de 80% das conexões possuem a qualidade máxima representada pela métrica ETX.

A partir de tais informações é possível projetar redes maiores a fim de testar a escalabilidade dos algoritmos, mas ainda mantendo as características peculiares à uma rede *mesh*.

Devido à ausência de trabalhos para caracterizar a distribuição da métrica ETX em uma rede de malha sem fio, outras referências foram consultadas para a continuidade deste trabalho, apesar de um estudo para a caracterização das variáveis aleatórias de redes em malha sem fio estar fora do escopo do mesmo.

Com o auxílio da ferramenta estatística R (R Core Team, 2013), a definição

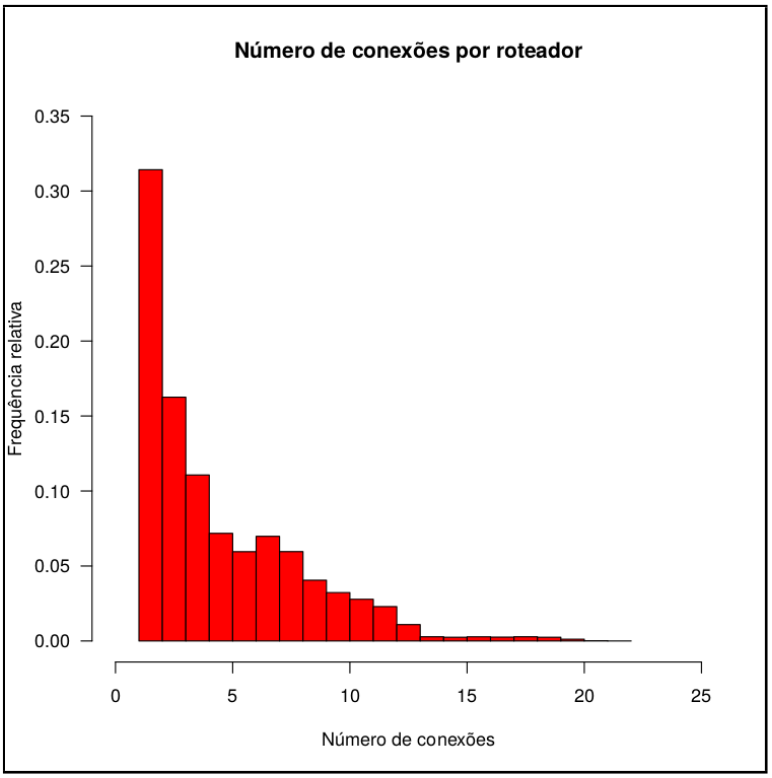

Gráfico 1 – Número de conexões por roteador.
Fonte: Autoria própria.

do grau dos nós foi melhor caracterizada pela distribuição de Poisson (OS; ENYI, 1959; OS; ENYI, 1960; OS; ENYI, 1962) utilizando o parâmetro lambda=4.809404. Para a relação de pesos dos enlaces, o pacote Fitdistrplus (DELIGNETTE-MULLER; DUTANG, 2015), pertencente ao R indicou a distribuição lognormal como a que melhor aproximou os dados coletados em uma rede real de acordo os dados de Wollenberg (2012).

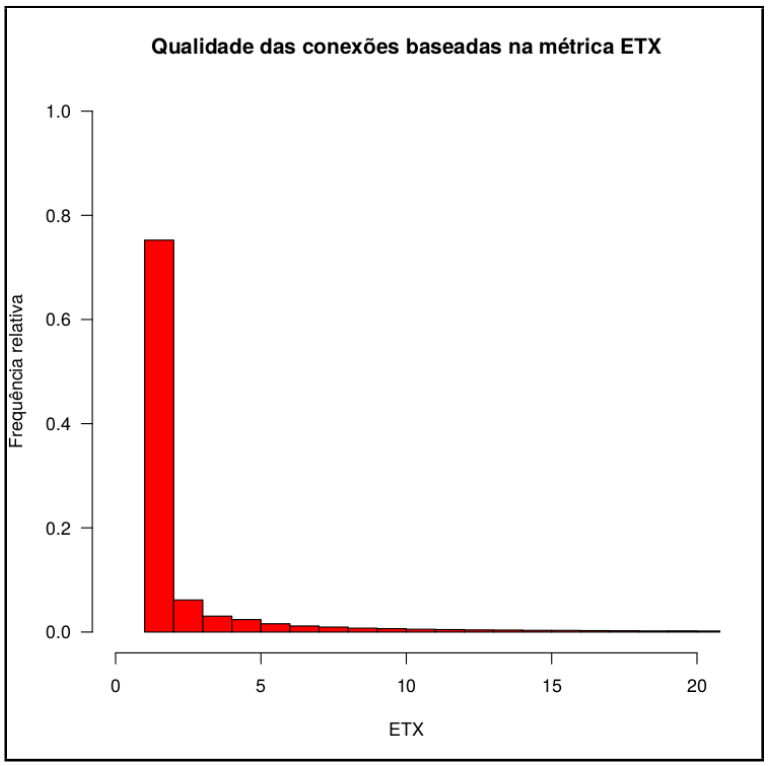

Gráfico 2 – Qualidade das conexões de acordo com a métrica ETX.
Fonte: Autoria própria.

5 RESULTADOS

As adaptações aplicadas na árvore Van Emde Boas propostas neste trabalho permitiram a utilização como fila de prioriodade no Algoritmo de Dijkstra uma estrutura de dados desenvolvida para outras aplicações.

No problema dos caminhos mínimos de origem única, varios vértices podem possuir o mesmo custo associado, apesar de utilizarem caminhos distintos. Dado este cenário, os valores inseridos na fila de prioriodade (relacionado ao custo dos caminhos) também podem apresentar uma redundância. Oringinamente a VEB, se utilizada no Algoritmo de Dijkstra, não aplica um tratamento eficaz para este problema.

O tratamento proposto neste trabalho para a inserção de chaves duplicadas permitiu armazenar apenas uma representação da chave, porém podendo a associar a diversos vértices do grafo. A identificação dos vértice relacionados a chave se dá por meio de uma lista com o identificador de cada vértice envolvido. A incorporação do nó da lista que relaciona a chave ao vértice na estrutura do próprio vértice envolvido permitiu executar os procedimentos inerentes a lista utilizada pela fila de prioriodade (pesquisa, inserção e remoção) a um custo constante.

Apesar desta alteração já permitir a utilização da VEB como fila de priorioda no Algoritmo de Dijkstra em muitos contextos, em uma rede de malha sem fio, o *hardware* em que o Algoritmo de Dijkstra será executado possui limitações quanto ao seu poder de processamento e armazenamento de dados. Para reduzir o espaço de memória utilizada pela estrutura, a implementação de uma árvore Van Emde Boas que alocasse dinamicamente seus nós se fez necessária.

Possuindo as estruturas aptas a serem utilizadas como fila de prioridade no Algoritmo de Dijkstra e tendo como base os dados apresentados por Wollenberg (2012), pôde-se avaliar o comportamento das filas de prioridade em grafos que representam características reais de uma rede de malha sem fio. Com base neste cenário projetado, resultados podem ser observados.

Considerando primeiramente a altura que as filas de prioridades podem assumir durante a execução do Algoritmo de Dijkstra como um dos indicativos de seu desempenho no pior caso, tem-se como esperado a VEB apresentando uma altura constante de 4 unidades, independente do número de itens nela contido.

Para o heap binário assume-se uma altura relativa a $\Theta(\log_2(|V|))$ por ser organizado como uma árvore binária quase completa. Quanto à AVL espera-se uma altura pouco acima da apresentada pelo heap binário, mas ainda na ordem de $(\log_2(|V|))$, sendo limitada superiormente por uma constante positiva, visto ser uma árvore balanceada mas não necessariamente completa.

Os resultados esperados em relação à altura das estruturas de dados podem

ser obtidos com base nos dados de uma rede *mesh* (OPENNET, 2015). O Gráfico 3 apresenta as alturas das filas de prioridade baseadas na topologia descrita pelas amostras da rede. O eixo x indica o índice dos registros da rede e o eixo y se refere à altura apresentada pelas estruturas para cada um destes registros.

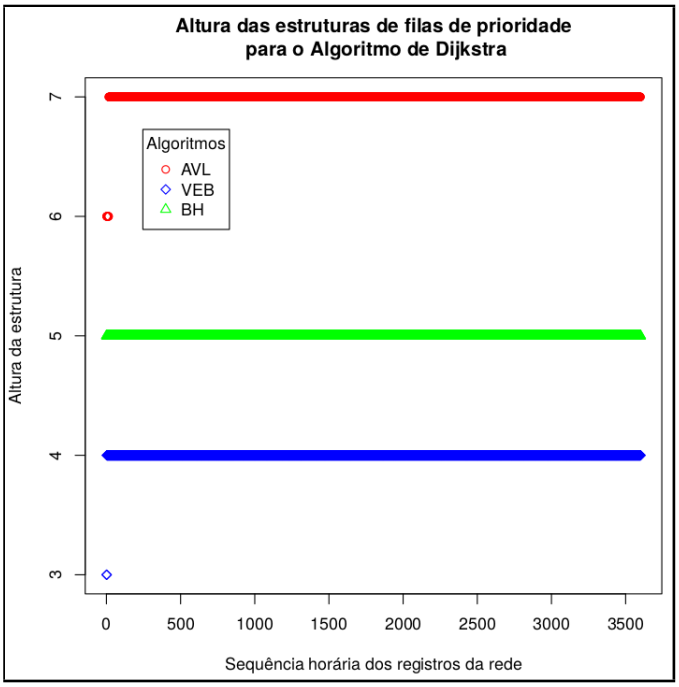

Gráfico 3 – Altura das filas de prioridade ao longo do tempo
Fonte: Autoria própria

A fim de verificar a escalabilidade das filas de prioridade, foram feitos testes em grafos completos (onde $|A| = (|V|\dot{|}V-1|)$) e em grafos que representam a configuração de uma rede *mesh*. Tais grafos, para fins de simplificação, serão denominados grafos *mesh* ou grafos esparsos. Tais testes corroboram o respaldo teórico que indica tempo constante para as operações de fila de prioridade da Van Emde Boas no contexto de uma rede de malha sem fio. Estes testes foram realizados em grafos com o número de vértices variando de 100 a 5000, crescendo a uma taxa de 100 vértices por amostra.

O Gráfico 4 apresenta o número de níveis que as filas de prioridade podem necessitar durante a execução do Algoritmo de Dijkstra em grafos completos com arestas valoradas de acordo com uma distribuição uniforme. O eixo x indica o crescimento do grafo baseando-se no número de nós e o eixo y o número de níveis apresentados pela estrutura. Nota-se que o padrão teórico esperado manteve-se nos resultados obtidos, uma vez que o heap binário apresentou fielmente uma altura na ordem de $\Theta(\log_2(|V|))$, a AVL uma altura pouco acima de $\log_2(|V|)$ e a VEB apresentando uma altura constante mesmo com o crescimento dos grafos.

Para grafos esparsos com arestas valoradas de acordo com as distribuições de probalidade descritas na Seção 4.5, o heap binário e a VEB mantiveram suas

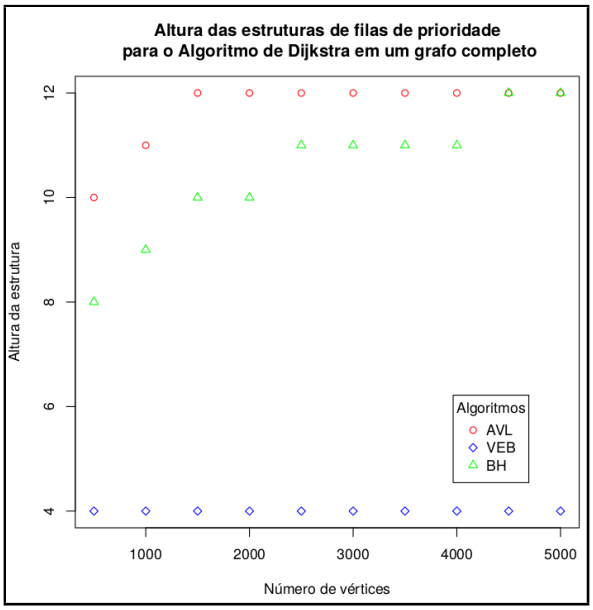

Gráfico 4 – Altura das filas de prioridade em grafos completos
Fonte: Autoria própria

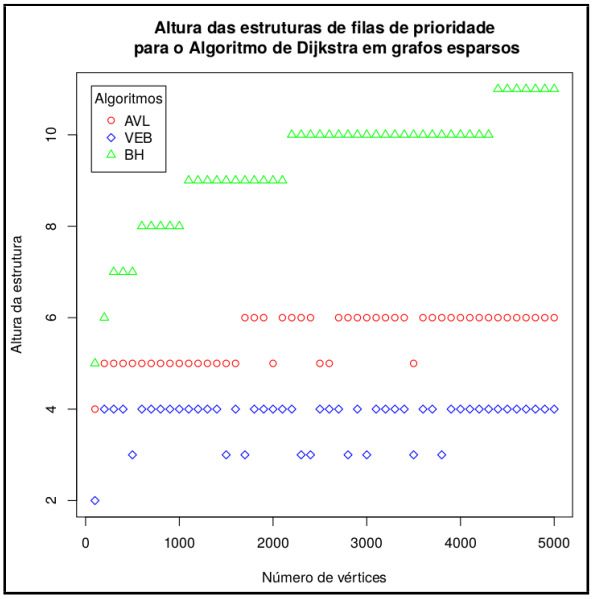

Gráfico 5 – Altura das filas de prioridade grafos esparsos
Fonte: Autoria própria

características. A AVL porém, apresentou uma significativa diminuição no número de níveis necessários, como visto no Gráfico 5. A característica apresentada pela AVL se deve ao fato da estrutura encadear lateralmente nós com o mesmo valor chave, acumulando sobre uma mesma chave vários nós, e assim não necessitando de mais níveis para organizá-los.

Esta característica não é facilmente notada em grafos completos se comparado com grafos esparsos devido ao maior intervalo de valores para as prioridades inseridas nas filas de prioridades, criado a partir do maior número de combinações de valores gerados em função dos custos das arestas.

O Gráfico 6 apresenta o número máximo de itens da AVL que possuem a mesma prioridade, além do número médio de nós para cada chave inserida na fila de prioridade durante a execução do Algoritmo de Dijkstra. Devido ao baixo número de arestas apresentado por uma rede *mesh* aliado aos baixos custos associados a elas, o número de combinações de valores se torna restrito, aumentando a repetição de valores associados aos vértices.

O número de níves necessários às filas de prioridade é um indicativo de seu comportamento com a escalabilidade da rede. Porém, para resultados mais práticos, adotou-se também o número de ciclos do processador necessários para a execução do algoritmo. Para identificar possíveis pontos de otimização, foram monitoradas as principais operações de uma fila de prioridade, ExtraiMínimo e DecrementaChave, além dos recursos utilizados na execução de todo o Algoritmo de Dijkstra.

Inicialmente, o Gráfico 7 ilustra os ciclos necessários às filas de prioridade

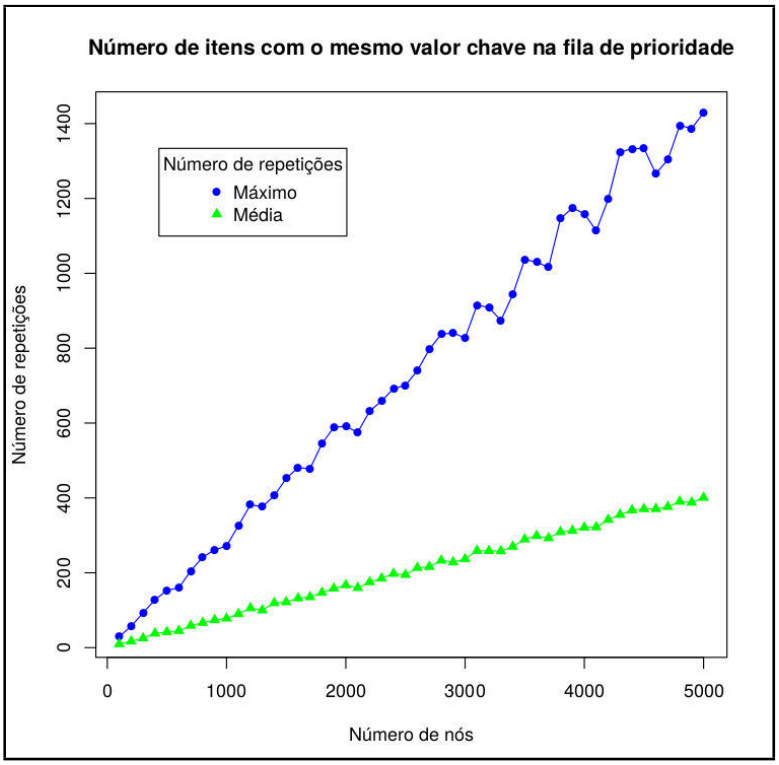

Gráfico 6 – Número de itens partilhando a mesma prioridade na fila de prioridade durante o cálculo de caminhos mínimos em um grafo *mesh*.
Fonte: Autoria própria

durante a execução do procedimento ExtraiMínimo. Os resultados obtidos tendo como base cada registro da rede reportada por Wollenberg (2012). Assim como na análise assintótica de pior caso, o heap binário também apresentou o pior desempenho neste procedimento, respeitando o ponto n_0 no qual este resultado é valido, devido à media de 175 vértices apresentada pelos grafos.

Abordou-se também o desempenho do procedimento DecrementaChave. Assim como no procedimento ExtraiMínimo, os resultados são respaldados pela análise teórica, apresentando o heap binário como o algoritmo de melhor desempenho para esta função. O Gráfico 8 apresenta os ciclos do processador necessários para que os algoritmos executem a função DecrementaChave nos grafos representados pela rede.

A fim de avaliar o desempenho total da execução do Algoritmo de Dijkstra, o Gráfico 9 apresenta os resultados obtidos. Devido ao número reduzido de vértices e, inferior ao intervalo no qual a AVL deixa de ser a melhor opção, os resultados práticos apresentados se mostram condizentes com a complexidade dos algoritmos apresentadas na avaliação teórica das Seções 4.2, 4.3 e 4.4.

Apesar dos resultados apresentarem o comportamento das filas de prioridade em uma rede de malha sem fio real, a escala desta rede permite identificar a melhor estrutura de dados apenas para esta configuração específica. A fim de avaliar o desempenho em redes de variadas escalas, foram produzidos resultados para grafos

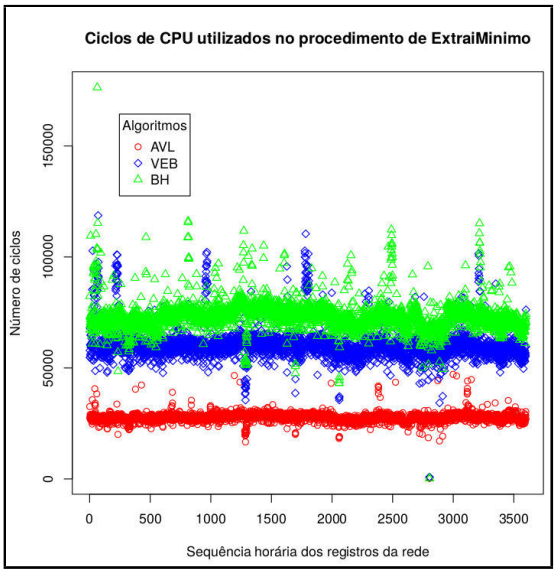

Gráfico 7 – Ciclos de CPU utilizados pelo procedimento ExtraiMínimo ao longo do tempo
Fonte: Autoria própria

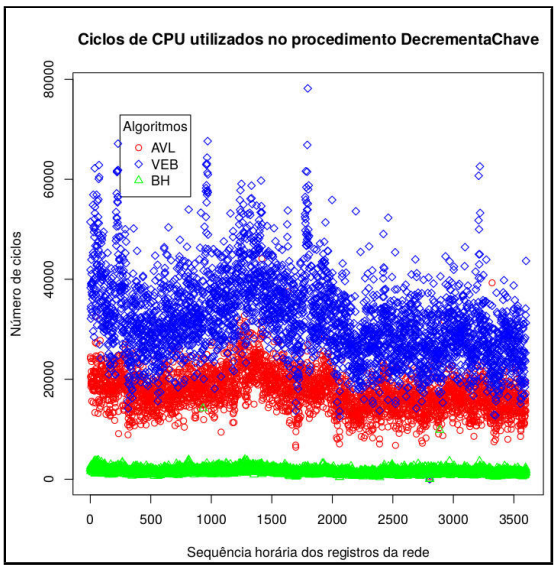

Gráfico 8 – Ciclos de CPU utilizados no procedimento Decrementa-Chave ao longo do tempo
Fonte: Autoria própria

maiores construídos de acordo com as distribuições de Poisson e lognormal.

O Gráfico 10 apresenta os ciclos de CPU necessários ao procedimento Extrai-Mínimo obtidos em grafos esparsos. AVL e VEB apresentam um desempenho similar, além de ambas necessitarem de menos ciclos para esta operação comparadas com o heap binário, sendo ambos os resultados respaldados pela análise assintótica dos algoritmos para esta operação.

A operação DecrementaChave também foi avaliada neste contexto, apresentando a AVL com um resultado significativamente pior. Este resultado é explicado pelo aumento do número de itens da fila de prioridade que compartilham do mesmo valor chave como visto no Gráfico 6. Diferentemente da VEB e do heap binário, a AVL os organizam em listas, transformando o custo anteriormente logarítmico das operações sobre eles em uma complexidade linear. O Gráfico 11 ilustra o desempenho das filas de prioridade em grafos *mesh*.

Com base nos resultados obtidos pelos algoritmos quanto as duas operações de filas de prioridade apresentadas, identifica-se o heap binário e VEB como os melhores desempenhos para a operação de DecrementaChave. Para a operação Extrai-Mínimo, AVL e VEB se mostram mais eficientes. Apesar da VEB possuir a alocação dinâmica de nós durante a execução do Algoritmo de Dijkstra, a estrutura apresentou resultados satisfatórios para ambos os procedimentos de fila de prioridade.

A fim de complementar o resultado dos desempenhos apresentados pelos algoritmos para as operações de forma isolada, buscou-se também avaliar os ciclos de processamento para toda a execução do Algoritmo de Dijkstra em cenários de grafos *mesh*. O Gráfico 12 ilustra os resultados obtidos.

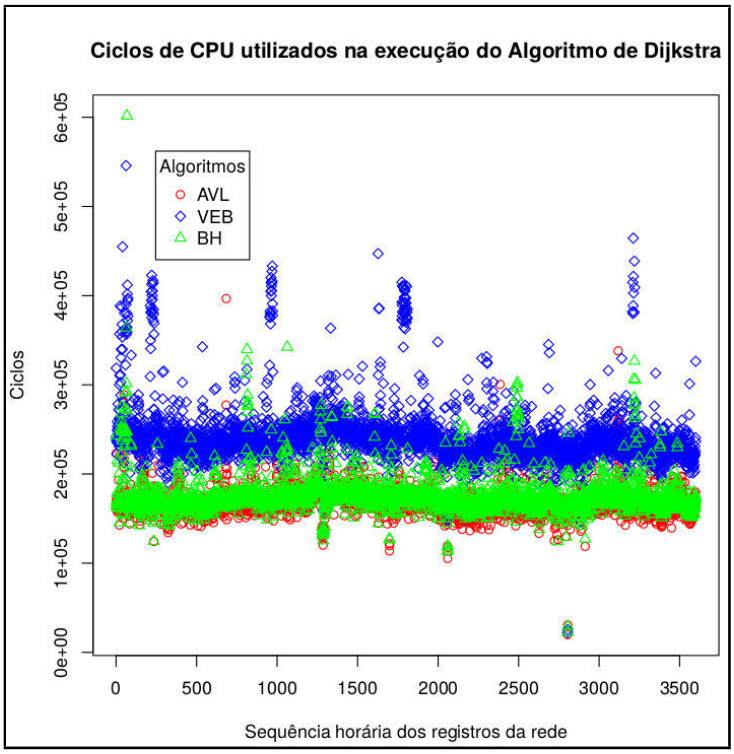

Gráfico 9 – Ciclos de CPU utilizados pelo Algoritmo de Dijkstra ao longo do tempo
Fonte: Autoria própria

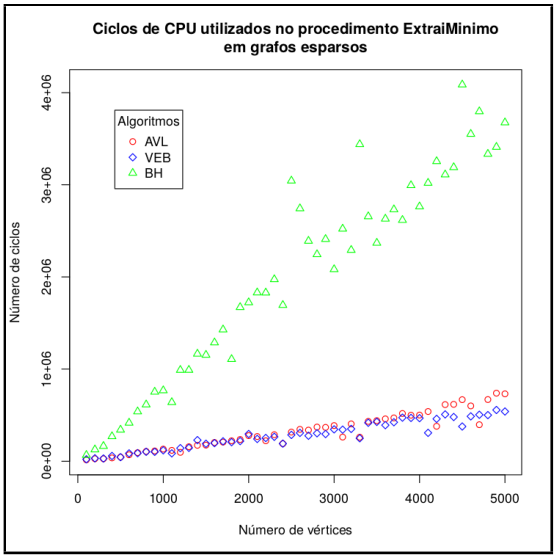

Gráfico 10 – Ciclos de CPU utilizados pelo procedimento ExtraiMínimo em grafos *mesh*
Fonte: Autoria própria

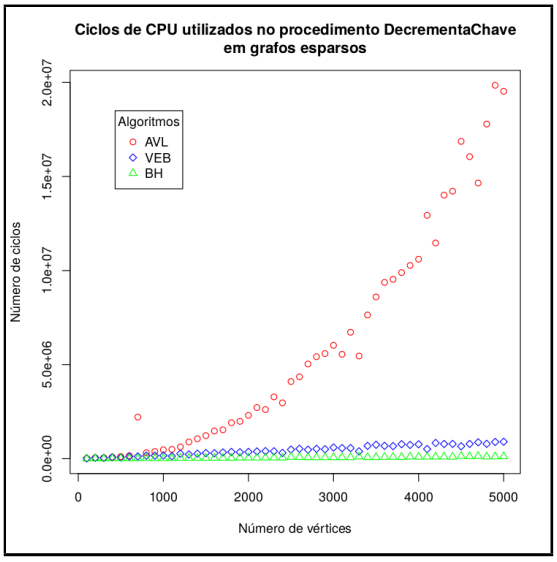

Gráfico 11 – Ciclos de CPU utilizados pelo procedimento Decrementa-Chave em grafos *mesh*
Fonte: Autoria própria

O Gráfico 12 apresenta a AVL como a estrutura menos otimizada neste cenário, este resultado é justificado pelo seu desempenho na operação DecrementaChave, como visto no Gráfico 11.

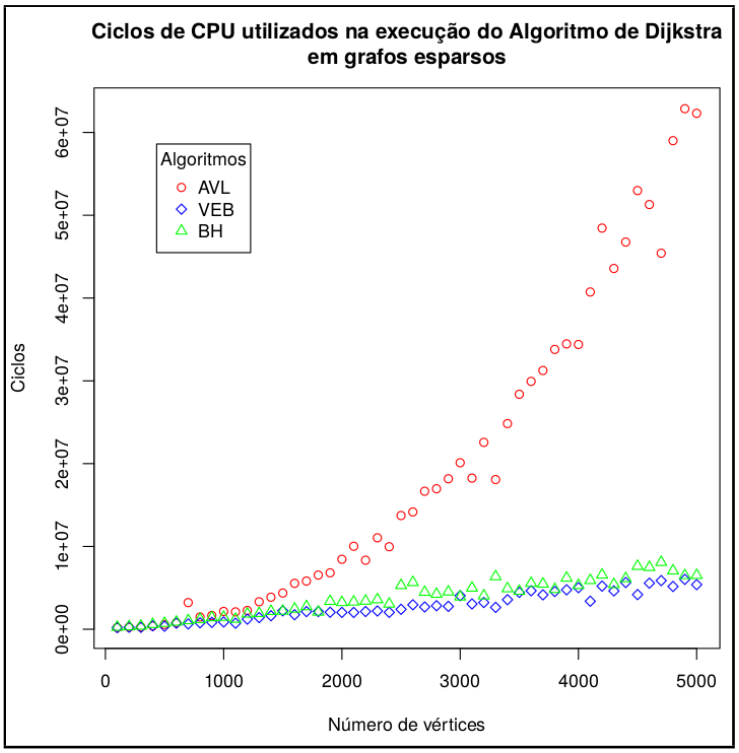

Gráfico 12 – Ciclos de CPU utilizados pelo Algoritmo de Dijkstra em função do crescimento de um grafo *mesh*

Fonte: Autoria própria

6 CONCLUSÃO

Este trabalho apresenta uma análise acerca das filas de prioridade mais adequadas para o Algoritmo de Dijkstra em um protocolo proativo em redes de malha sem fio. A partir da análise assintótica da fila de prioridade utilizada pelo protocolo OLSRD pode-se buscar na literatura outras estruturas de dados que teoricamente poderiam melhorar o desempenho do Algoritmo de Dijkstra neste contexto.

A Van Emde Boas e o heap binário mostram-se eficientes teoricamente comparadas à AVL, estrutura utilizada pelo protocolo. A análise assintótica destas estruturas bem como testes práticos realizados com elas mostram que a operação DecrementaChave é mais eficiente na estruturas VEB e heap binário, enquanto a operação ExtraiMínimo é mais eficiente na AVL e VEB.

A partir da análise dos valores inseridos na fila de prioridade durante a execução do Algoritmo de Dijkstra utilizando dados de uma rede de malha sem fio real e projeções estatísticas baseadas nestes dados, foi identificado que grande parte dos itens inseridos na fila possuem prioridades repetidas entre si. Até 25% dos elementos da fila de prioridade podem compartilhar uma mesma prioridade em específico e cada valor distinto de prioridade pode estar associado em média a 12% dos itens contidos na fila.

O grande número de prioridades repetidas identificadas em redes de malha sem fio influencia o desempenho da filas de prioridades utilizadas pelo Algoritmo de Dijkstra. Apesar de estruturas do tipo lista possuirem tradicionalmente uma complexidade linear em seu procedimento de busca, graças à inclusão dos nós do tipo lista, utilizados pela Van Emde Boas, à estrutura de cada vértice do grafo, foi possível a busca por esses itens nas listas de cada nó VEB em um tempo constante.

O desempenho da AVL implementada no protocolo OLSRD poderia ser melhorado caso a ordem dos itens da lista de elementos de mesma prioridade fosse irrelevante, a exemplo do que foi feito na implementação da árvore Van Emde Boas neste trabalho. Ainda assim, o desempenho da AVL não deve superar o desempenho da Van Emde Boas devido à complexidade constante das operações de fila de prioridade na VEB, que apontam para um desempenho melhor que da AVL com suas operações de complexidade logarítmica.

Registros apresentados pelos testes práticos também confirmam a altura constante da VEB durante a execução do Algoritmo de Dijkstra, independente do tamanho da rede em questão, desde que as chaves a serem inseridas na estrutura respeitem o intervalo u de valores válidos, sendo este intervalo delimitado por uma constante pelo protocolo durante sua execução. Esta constatação é fundamental para permitir a execução das operações da fila em tempo $\log_2 \log_2(u)$. Este desempenho permite

teoricamente uma complexidade linear para a execução do Algoritmo de Dijkstra.

Este desempenho linear, porém, não foi identificado durante os testes práticos realizados com a VEB. Este resultado pode ser justificado pela escala utilizada para a construção dos grafos e/ou pelo consumo de recursos adicionais relacionados à alocação dinâmica da estrutura durante a execução do Algoritmo, recursos estes, que não foram necessários às demais filas estudadas.

Em uma avaliação geral, o heap binário juntamente com a VEB apresentaram os melhores desempenhos para grafos esparsos. O heap porém, apresentou um pequeno incremento em seu número de ciclos se comparado a VEB, tal incremento é justificado pelo seu desempenho no procedimento ExtraiMínimo. Este acréscimo porém, não apresentou perdas significativas ao seu desempenho se comparado ao procedimento DecrementaChave para a AVL, ponto fraco apresentado pela árvore. Para pequenas redes, menores que 800 nós, a árvore AVL apresentou o melhor desempenho devido ao número de execuções do procedimento DecrementaChave não ser significativamente grande nestes grafos.

Apesar de não apresentar um desempenho linear para o Algoritmo de Dijkstra, a Van Emde Boas manteve um bom desempenho nas duas operações de fila de prioridade avaliadas, mesmo necessitando de recursos adicionais para o processo de alocação dinâmica de seus nós durante sua execução, mostrando-se a fila de prioridade com o melhor desempenho prático para redes *mesh* a partir de 800 nós.

A partir dos resultados obtidos pelas duas filas de prioridades sugeridas e analisando a compatibilidade atual dessas estruturas com a atual implementação do Algoritmo de Dijkstra do protocolo OLSRD, foi escolhido o heap binário como a melhor fila de prioridade para protocolo atualmente.

Apesar dos melhores resultados da VEB, os recursos adicionais exigidos por ela resultariam em alterações drásticas na organização do protocolo. O heap binário por sua vez, obteve um desempenho pouco inferior ao da VEB para redes a partir de 800 nós e um desempenho melhor para redes menores, obtendo um desempenho similar a AVL para estas redes. Além de seu desempenho apresentado, o heap binário mantêm algumas características da AVL, resultando em uma adaptação mais natural da fila de prioridade no Algoritmo de Dijkstra do OLSRD.

A implementação do heap binário no protocolo mostrou-se estável em testes realizados no emulador OLSR_switch. Após a validação dos testes e a conclusão acerca da viabilidade do uso da fila de prioridade no protocolo, foi proposto à comunidade responsável pelo OLSRD a substituição da fila de prioridade atualmente presente no protocolo pelo uso de um heap binário. A proposta foi aceita pela comunidade e atualmente encontra-se em fase de teste.

Os resultados deste trabalho também foram divulgados no Forúm de Técno-

logia em Software Livre 2015 (FTSL, 2015)

6.1 TRABALHOS FUTUROS

A partir dos resultados apresentados por este estudo, alguns trabalhos podem ser executados para dar continuidade à avaliação de desempenho do procedimento de cálculo dos caminhos mínimos necessário ao protocolo de roteamento de redes de malha sem fio. Dentre alguns tópicos que complementariam assuntos não abordados por este trabalho, podem ser destacados:

- Implementação da VEB sem o consumo adicional de recursos de memória durante a execução do Algoritmo de Dijkstra: uma característica peculiar apresentada pela Van Emde Boas se comparada com as demais filas estudadas foi a necessidade da alocação sob demanda de seus nós durante a execução do Algoritmo de Dijkstra. Devido ao espaço de memória variável necessário ao atributo *cluster* contido em cada nó, dependente do intervalo de valores associados a ele durante a execução do algoritmo, a prévia alocação dos nós da estrutura se mostra comprometida. Como uma estrutura tc_entry[1] é criada apenas uma vez para cada roteador da rede, alocar previamente o nó VEB juntamente com o nó tc_entry contribuiria para uma melhora no desempenho geral do algoritmo, visto que a estrutura seria alocada apenas uma vez independente do número de chamadas ao Algoritmo de Dijkstra.

- Avaliação das filas de prioridade em algoritmos incrementais: apesar dos resultados apresentados pelas filas de prioridade no Algoritmo de Dijkstra contribuírem para o desempenho do protocolo, estudos apresentaram melhorias de desempenho de redes de malha sem fio utilizando algoritmos incrementais como base para o cálculo de caminhos mínimos. Apesar do método de cálculo ser fortemente baseado no Algoritmo de Dijkstra, algumas peculiaridades apresentadas pelos algoritmos incrementais podem alterar características importantes acerca dos dados inseridos na fila de prioridade utilizada, como por exemplo a proporção de chaves repetidas. Combinar os pontos fortes dos estudos sobre filas de prioridade e algoritmos incrementais pode potencializar o desempenho de protocolos de redes de malha sem fio.

- Avaliação de desempenho das filas de prioridades em outros exemplos de redes *mesh*: devido a escassez de trabalhos reportando as configurações

[1] Descrita na Seção 3.7

topológicas de uma rede de malha sem fio baseadas na métrica ETX, foi necessário a criação de cenários baseados em distribuições de probabilidade que representam as características de uma WMN apresentada no estudo de Wollenberg (2012). Idealmente, a avaliação da escalabilidade das filas de prioridade poderia ser aplicada tendo como base dados de redes maiores. Um estudo futuro para a avaliação das estruturas utilizando outros exemplos de redes *mesh* traria mais credibilidade aos resultados obtidos.

- Consolidação das otimizações implementadas no protocolo OLSRD: as alterações propostas à comunidade mantenedora do protocolo OLSRD foram aprovadas pela comunidade, o que resultou na criação de um *branch* com a versão proposta no repositório oficial do protocolo visando torná-la a opção padrão na versão 2 do projeto. Com a versão proposta incorporada ao projeto, o desempenho do protocolo poderá ser aprimorado a partir da implementação dos trabalhos descritos nos itens acima.

REFERÊNCIAS

ADELSON-VELSKY, G. G.; LANDIS, E. M. An Algorithm for the Organization of Information. **Doklady Akademii Nauk USSR**, Russia, v. 146, n. 2, p. 263–266, 1962.

AWMN. **ATHENS WIRELESS METROPOLITAN NETWORK**. 2014. Disponível em: <http://www.awmn.net>. Acesso em: 29 mai. 2014.

BOAS, P. van E. Preserving order in a forest in less than logarithmic time and linear space. **Proceedings of the 16th Annual Symposium on Foundations of Computer Science Algorithms**, v. 6, n. 3, p. 80–82, 1977.

BONDY, J.; MURTY, U. **Graph Theory**: Graduate Texts in Mathematics. Londres, Inglaterra: Springer, 2008.

CERF, V. G.; KHAN, R. E. A protocol for packet network intercommunication. **IEEE Transactions on communications**, [S.l.], v. 22, p. 637–648, 1974.

CORMEN, T. et al. **Introduction to algorithms**. Cambridge, EUA: The MIT Press, 2009.

COUTO, D. S. J. de. **High-Throughput Routing for Multi-Hop Wireless Networks**. Tese (Doctor of Philosophy) - Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, EUA, 2004.

DELIGNETTE-MULLER, M. L.; DUTANG, C. Fitdistrplus: An R package for fitting distributions. **Journal of Statistical Software**, [S.l.], v. 64, n. 4, p. 1–34, 2015. Disponível em: <http://www.jstatsoft.org/v64/i04/>.

DIJKSTRA, E. W. A note on two problems in connection with graphs. **Numerische Mathematik**, [S.l.], v. 1, n. 1, p. 269–271, 1959.

ERCIYES, k. **Distributed Graph Algorithms for Computer Networks**. Londres, Inglaterra: Springer, 2013.

FRANGOUDIS, P.; POLYZOS, G.; KEMERLIS, V. Wireless community networks: an alternative approach for nomadic broadband network access. **Communications Magazine, IEEE**, [S.l.], v. 49, n. 5, p. 206–213, 2011.

FREIFUNK. **What is Freifunk?**. 2014. Disponível em: <http://berlin.freifunk.net/index_en/>. Acesso em: 29 mai. 2014.

FREIFUNK FIRMWARE: A non-commercial initiative for free wireless networks. Versão 0.1.2. Alemanha: Freifunk Berlin. 2015. Disponível em: <http://freifunk.net/>.

FÓRUM DE TECNOLOGIA EM SOFTWARE LIVRE, 7., 2015, Curitiba. **Palestra...** Curitiba: UTFPR, 2015. Disponível em: <http://ftsl.org.br/grade/?id=cfcd208495d565ef66e7dff9f98764da#>. Acesso em: 10 Set. 2015.

IEEE. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. EUA, 2012. 2695 p.

INTERNET ENGINEERING TASK FORCE. **RFC 3626**: Optimized Link State Routing Protocol (OLSR). Inria, França, 2003.

KNUTH, D. E. **The Art of Computer Programming**: Sorting and searching. Boston, EUA: Person Education, 1998.

LINDHORST, T.; LUKAS, G.; NETT, E. Wireless mesh network infrastructure for industrial applications. **Emerging Technologies & Factory Automation (ETFA)**, Cagliari, Italia, v. 18, p. 1–8, 2013.

MORRIS, J. M. Traversing binary trees simply and cheaply. **Information Processing Letters**, [S.l.], n. 9, p. 197-200, 1979.

NORTEL. textbfSolutions Breaf: Wireless mesh network. EUA, 2006. 8 p.

OBERHAUSER, G.; SIMHA, R. Fast data structures for shortest path routing: a comparative evaluation. **IEEE International Conference on Seattle**. Seattle, EUA, v. 3, p. 1597–1601, 1995.

OLSR.ORG. **Projects - OLSR.org Wiki**. 2015. Disponível em: <http://www.olsr.org/mediawiki/index.php/Projects>. Acesso em: 13 Set. 2015.

OPEN-MESH. **B.A.T.M.A.N. protocol concept**. 2014. Disponível em: <http://www.open-mesh.org/projects/open-mesh/wiki/BATMANConcept>. Acesso em: 23 Set. 2014.

OPENNET. **Opennet-initiative**. 2015. Disponível em: <https://wiki.opennet-initiative.de/wiki/Hauptseite>. Acesso em: 23 Abr. 2015.

OPENWRT: OpenWrt is described as a Linux distribution for embedded devices. Versão 15.05. [S.l]: OpenWrt. 2015. Disponível em <https://openwrt.org/>.

OS, P. E.; ENYI, A. R. On random graphs. Publicationes Mathematicae, Hungria, 1959.

____On the evolution of random graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences, Hungria, 1960.

____On the strength of connectedness of a random graph. Acta Mathematica Scientia Hungary, Hungria, 1962.

QUEIROZ, S. J. B. de. **Avaliação de Roteamento Incremental em Redes em Malha Sem Fio Baseadas em 802.11**. Dissertação (Mestrado em Informática) - Departamento de Ciência da Computação, Universidade Federal do Amazonas, Manaus, 2009.

R: A Language and Environment for Statistical Computing. Versão 3.2. Austria: R Core Team. 2013. Disponível em: <http://www.R-project.org/>.

RAMALINGAM, G.; REPS, T. An incremental algorithm for a generalization of the shortest-path problem. **Journal of Algorithms**, [S.l.], v. 21, n. 2, p. 267–305, 1996.

SEATTLE WIRELESS. **Free Wifi Networking**. 2014. Disponível em: <https://openwrt.org/>. Acesso em: 29 mai. 2014.

SEDGEWICK, R.; FLAJOLET, P. **An introduction to the analysis of algorithms**. Courier in Westford, EUA: Pearson Education, 2013.

SEDGEWICK, R.; WAYNE, K. **Algorithms**. Courier in Westford, EUA: Pearson Education, 2011.

SIRIS, V.; TRAGOS, E.; PETROULAKIS, N. Experiences with a metropolitan multiradio wireless mesh network: design, performance, and application. **Communications Magazine, IEEE**, [S.l.], v. 50, p. 128–136, 2012.

TONNESEN, A. **Impementing and extending the Optimized Link State Routing Protocol**. Dissertação (Mestrado em Informática) - UniK University Graduate Center - Department of Informatics, University of Oslo, Oslo - Noruega, 2004.

WIKIMEDIA COMMONS. **Threaded tree**. 2015. Disponível em: <https://commons.wikimedia.org/wiki/File:Threaded_tree.svg>. Acesso em: 22 mai. 2015.

WOLLENBERG, T. Performance measurement study in a wireless olsr-etx mesh network. **IFIP Wireless Days**, Dublin, Irlanda, p. 1–7, 2012.

YU, Z.; XU, X.; WU, X. Application of wireless mesh network in campus network. **Communication Systems, Networks and Applications (ICCSNA)**, Hong Kong, China, v. 1, p. 245–247, 2010.

APÊNDICE A - Procedimentos utilizados pela árvore AVL

O procedimento de inserção em uma AVL utilizado pelo protocolo é apresentado no Quadro 16. O procedimento recebe como parâmetro o nó gerenciador da árvore do tipo *avl_tree*, o nó a ser inserido e um indicador da utilidade da árvore (grafo ou fila de prioridade).

Inicialmente o nó a ser inserido tem seus campos configurados, sendo a princípio balanceado e único na árvore (linhas 17 e 18). Caso a estrutura esteja vazia, o nó é configurado como o único elemento da árvore e os campos do nó gerenciador são atualizados. Caso contrário a posição do nó de acordo com sua chave é encontrada na árvore (linha 28).

A partir da posição do nó na árvore, caso já exista elementos com a mesma prioridade, o nó inserido deverá ser posicionado na última posição da lista (linhas 31 a 34). Esta operação se mostrou a de maior complexidade no procedimento, podendo apresentar uma complexidade na ordem de $O(n)$ no pior caso, onde n é o número de itens na lista, já que esta lista pode conter todos os nós da árvore caso sejam inseridos apenas nós com a mesma prioridade. Porém, a inserção da nova chave na última posição da lista se mostra apenas uma estratégia da implementação visto que a posição do nó na lista não influência sua prioridade na árvore. A inserção do nó na primeira posição da lista permite executar esta operação em $O(1)$.

Uma comparação de valores é feita entre o nó a ser inserido e o nó que se encontra na posição onde será feita a inserção. Caso as chaves sejam iguais (linhas 38 a 46), o item é inserido na lista (caso seja uma fila de prioridade). Caso a árvore seja utilizada para armazenar o grafo, o algoritmo retorna um erro, já que a repetição de um nó indicaria dois vértices com o mesmo identificador. As linhas 48 a 64 inserem o nó na subárvore à direita ou à esquerda de acordo com o balanceamento do nó pai. As linhas 66 a 74 inserem o nó à esquerda do nó pesquisado, caso o chave possua um valor inferior. O procedimento auxiliar *post_insert* corrige o balanceamento da árvore, executando rotações se necessário.

```
1  int
2  avl_insert(struct avl_tree *tree, struct avl_node *new, int
       allow_duplicates)
3  {
4      struct avl_node *node;
5      struct avl_node *last;
6      int diff;
7      int tam=0;
8
9      new->parent = NULL;
10
11     new->left = NULL;
```

```
12      new->right = NULL;
13
14      new->next = NULL;
15      new->prev = NULL;
16
17      new->balance = 0;
18      new->leader = 1;
19
20      if (tree->root == NULL) {
21          tree->root = new;
22          tree->first = new;
23          tree->last = new;
24          tree->count = 1;
25          return 0;
26      }
27
28      node = avl_find_rec(tree->root, new->key);
29      last = node;
30
31      while (last->next != NULL && last->next->leader == 0){
32          last = last->next;
33          tam++;
34      }
35
36      diff = (new->key - node->key);
37
38      if (diff == 0) {
39          if (allow_duplicates == AVL_DUP_NO)
40          return -1;
41
42          new->leader = 0;
43
44          avl_insert_after(tree, last, new);
45          return 0;
46      }
47
48      if (node->balance == 1) {
49          avl_insert_before(tree, node, new);
50
51          node->balance = 0;
52          new->parent = node;
53          node->left = new;
```

```
54          return 0;
55      }
56
57      if (node->balance == -1) {
58          avl_insert_after(tree, last, new);
59
60          node->balance = 0;
61          new->parent = node;
62          node->right = new;
63          return 0;
64      }
65
66      if (diff < 0) {
67          avl_insert_before(tree, node, new);
68
69          node->balance = -1;
70          new->parent = node;
71          node->left = new;
72          post_insert(tree, node);
73          return 0;
74      }
75
76      avl_insert_after(tree, last, new);
77
78      node->balance = 1;
79      new->parent = node;
80      node->right = new;
81      post_insert(tree, node);
82      return 0;
83  }
```

Quadro 16 – Procedimento de inserção em uma árvore AVL
Fonte: Autoria própria

Tendo como parâmetros o nó gerenciador da AVL e o elemento a ser removido da árvore, o procedimento de remoção da AVL, ilustrado no Quadro 17, inicialmente verifica se o nó a ser removido esta incluído no corpo de uma lista de outros elementos com a mesma prioridade (linha 9), caso isto ocorra o procedimento o remove da lista em um tempo constante com o auxílio do procedimento *avl_remove*.

Caso o nó removido seja a cabeça de uma lista (linhas 12 a 39), o próximo nó assume seu lugar no início da lista, caso contrário o procedimento *avl_delete_worker* é invocado para efetuar a remoção do nó e corrigir o balanceamento da árvore.

```
1  void
2  avl_delete(struct avl_tree *tree, struct avl_node *node)
3  {
4      struct avl_node *next;
5      struct avl_node *parent;
6      struct avl_node *left;
7      struct avl_node *right;
8
9      if (node->leader != 0) {
10         next = node->next;
11
12         if (next != NULL && next->leader == 0) {
13             next->leader = 1;
14             next->balance = node->balance;
15
16             parent = node->parent;
17             left = node->left;
18             right = node->right;
19
20             next->parent = parent;
21             next->left = left;
22             next->right = right;
23
24             if (parent == NULL)
25                 tree->root = next;
26
27             else {
28                 if (node == parent->left)
29                     parent->left = next;
30
31                 else
32                     parent->right = next;
33             }
34
35             if (left != NULL)
36                 left->parent = next;
37
38             if (right != NULL)
39                 right->parent = next;
40         }
41
42         else
43             avl_delete_worker(tree, node);
44     }
45
46     avl_remove(tree, node);
47 }
```

Quadro 17 – Procedimento de remoção em uma árvore AVL
Fonte: Autoria própria

APÊNDICE B - Procedimentos utilizados pelo heap binário

O procedimento heapExtractMin, resposável pela remoção do nó x com a maior prioriodade no heap necessita redirecionar um novo nó para a raiz da estrutura, o nó escolhido é elemento da última posição do heap, que após ser inserido na raiz terá sua posição corrigida de acordo com sua chave.

Após a remoção do nó x, o procedimento para identificação do último elemento do heap (heapFindLastNode), ilustrado no Quadro 18, desenvolvido para a biblioteca do heap binário utilizada neste estudo executa a busca pelo elemento em um tempo na ordem de $O(\log_2 n)$. Tendo a estrutura do heap como parâmetro, o procedimento calcula, com o auxílio do procedimento heapPerfectLog2 (executado em tempo constante[2]) se o último nível da estrutura encontra-se cheio. Em caso afirmativo, o último elemento se encontrará no nó mais à direita da estrutura, caso contrário deve-se encontrar o nó mais à direita posicionado à esquerda do nó ainda assinalado como último. No pior caso o último elemento encontra-se no nó mais à direita da subárvore à esquerda do nó raiz, forçando o procedimento a executar uma verificação (linha 11) até a raiz do heap a partir do último nível da subárvore direita e posteriormente executar o caminho inverso, desta vez pela subárvore esquerda.

```
1  node* heapFindLastNode(heap *root){
2      node *aux = root->last_node;
3      unsigned int N = root->count+1;
4      if ( !heapPerfectLog2(N) ){
5          aux = root->root_node;
6          while (aux->right) {
7              aux = aux->right;
8          }
9      }
10     else if ( N % 2 == 0){
11         while(aux->parent->left == aux)
12             aux = aux->parent;
13         aux = aux->parent->left;
14         while(aux->right)
15             aux = aux->right;
16     }
17     return aux;
18 }
```

Quadro 18 – Procedimento para identificação do último nó presente no heap após a remoção de um elemento

Fonte: Autoria própria

O procedimento auxiliar para a inserção de um nó utilizado para identificar o nó pai do elemento a ser inserido utiliza o mesmo conceito apresentado anteriormente para o procedimento heapFindLastNode. A função heapFindParentInsertNode, ilustrada no Quadro 19, baseia-se no preenchimento total do último nível do heap para

[2] constante limitada pelo número de bits da variável como parâmetro

a busca do nó, o alvo porém é o pai do nó a ser inserido. Caso o heap esteja cheio, um novo nível deverá ser iniciado e o pai do novo nó será o item mais à esquerda na estrutura (linhas 4 a 9), caso contrário será o item mais à direita do penúltimo nível do heap a não possuir um dos filhos.

```
1  node* heapFindParentInsertNode(heap *root){
2      node *aux = root->last_node;
3      unsigned int N = root->count+1;
4      if ( !heapPerfectLog2(N) ){
5          aux = root->root_node;
6          while (aux->left) {
7              aux = aux->left;
8          }
9      }
10     else if ( N % 2 == 0){
11         while(aux->parent->right == aux)
12             aux = aux->parent;
13         if(!aux->parent->right)
14             return aux->parent;
15         aux = aux->parent->right;
16         while(aux->left)
17             aux = aux->left;
18     }
19     else{
20         aux = aux->parent;
21     }
22     return aux;
23 }
```

Quadro 19 – Procedimento para identificação do nó pai do próximo item a ser inserido no heap binário

Fonte: Autoria própria

Ambos os procedimentos, heapFindLastNode e heapFindParentInsertNode, executarão no máximo $2 \log_2 n$ operações (linhas 11 a 15 e 11 a 17 respectivamente), mantendo a classe assintótica logarítmica esperada para procedimentos em um heap binário.

O procedimento heapDecreaseKey ilustrado no Quadro 20 é o responsável pela atualização de chaves no heap binário. O procedimento possui como parâmetros a estrutura do heap, o nó a ser atualizado e o novo valor da chave do nó. Baseada nesses itens, a função compara a chave recebida com a prioridade do nó pai do elemento atualizado (linha 12). Caso a nova prioridade infrinja a propriedade do heap, a troca entre os nós pai e filho é efetuada (linhas 13 a 52). No pior caso, o nó atualizado percorrerá todos os níveis da estrutura efetuando uma troca de posição com seu relativo pai, o trecho de código das linhas 12 a 52, contendo 8 comparações, será executado no máximo $\log_2 n$ vezes.

```
void heapDecreaseKey(heap *root, node *heap_node, keyType key){
    heap_node->key = key;
    node *parent = heap_node->parent;
    node *left = heap_node->left;
    node *right = heap_node->right;
    if(!parent)
        return;
    if(parent->key > heap_node->key){
        if(root->last_node == heap_node)
            root->last_node = parent;
    }
    while(parent && (parent->key > heap_node->key)){
        if(parent->left == heap_node){
            heap_node->left = parent;
            heap_node->right = parent->right;
            if(heap_node->right)
                heap_node->right->parent = heap_node;
            heap_node->parent = parent->parent;
            if(heap_node->parent){
                if(heap_node->parent->left == parent)
                    heap_node->parent->left = heap_node;
                else
                    heap_node->parent->right = heap_node;
            }
            else
                root->root_node = heap_node;
        }
        else{
            heap_node->right = parent;
            heap_node->left = parent->left;
            if(heap_node->left)
                heap_node->left->parent = heap_node;
            heap_node->parent = parent->parent;
            if(heap_node->parent){
                if(heap_node->parent->left == parent)
                    heap_node->parent->left = heap_node;
                else
                    heap_node->parent->right = heap_node;
            }
            else
                root->root_node = heap_node;
        }
```

```
43          parent->left = left;
44          parent->right = right;
45          parent->parent = heap_node;
46          if(left)
47              left->parent = parent;
48          if(right)
49              right->parent = parent;
50          parent = heap_node->parent;
51          left = heap_node->left;
52          right = heap_node->right;
53      }
54  }
```

Quadro 20 – Procedimento para aumento de prioridade de um nó no heap binário
Fonte: Autoria própria

A inserção de uma chave no heap binário é executada pelo procedimento heapInsert, visto no Quadro 21. Dados a estrutura do heap e o elemento a ser inserido, o procedimento configura a um tempo constante os ponteiros do nó com o auxílio da função heapInitNode (linha 4). O caso base ocorre quando o elemento do heap encontra-se vazio e o nó inserido passa a ser a raiz e único elemento da estrutura. Caso isto não ocorra, inicia-se o processo para localizar a posição inicial do novo elemento com o auxílio do procedimento heapFindParentInsertNode (linha 11), a relação pai/filho é corrigida (linhas 12 a 18) e os dados globais da estrutura são atualizados (linhas 19 a 20). Após o nó ser anexado ao heap, sua localização deve ser atualizada de acordo com seu valor chave, sendo utilizado o procedimento heapDecreaseKey para tal fim.

O procedimento heapExtractMin é o responsável pela remoção do elemento de maior prioridade do heap binário de mínimo. O procedimento possui 3 tratamentos para a reconfiguração do heap após a remoção da raiz da estrutura: o heap passa a estar vazio, conter um elemento e possuir dois ou mais elementos. O Quadro 22 ilustra o procedimento de extração do item de maior prioriodade na Fila.

Após a remoção do elemento desejado, caso a estrutura não contenha nenhum outro nó (linha 7) esta informação será informada a estrutura gerenciadora do heap. Caso o heap possua um elemento o nó restante será definido simultaneamente como o raiz e ultimo nó da estrutura. O tratamento especial ocorre quando restam mais nós no heap. O último elemento da estrutura será posicionado na raiz do heap (linha 31) e um novo elemento deverá assumir seu lugar (linha 18). Para garantir a propriedade da relação pai e filho do heap binário, o novo nó raiz deve ser posicionado corretamente na estrutura de acordo com sua prioridade na fila. O procedimento heapIncreaseKey é o responsável pela correção do posicionado do nó raiz.

```
1  void heapInsert(heap *root, node *heap_node){
2
3      node *parent = NULL;
4      heapInitNode(heap_node);
5
6      if(!root->count){
7          root->root_node = root->last_node = heap_node;
8          root->count++;
9      }
10     else{
11         parent = heapFindParentInsertNode(root);
12         if(parent->left){
13             parent->right = heap_node;
14         }
15         else{
16             parent->left = heap_node;
17         }
18         heap_node->parent = parent;
19         root->count++;
20         root->last_node = heap_node;
21         heapDecreaseKey(root, heap_node, heap_node->key);
22     }
23 }
```

Quadro 21 – Procedimento de inserção de um novo elemento em um heap binário
Fonte: Autoria própria

```
node *heapExtractMin(heap *root){
    node *min_node = root->root_node;
    node *new_min = root->last_node;
    if(!min_node)
        return NULL;
    root->count--;
    if(root->count == 0){
        root->last_node = root->root_node = NULL;
    }
    else if(root->count == 1){
        root->last_node = root->root_node = new_min;
        new_min->parent = NULL;
    }
    else{
        if(new_min->parent->left == new_min){
            new_min->parent->left = NULL;
            root->last_node = new_min->parent;
            root->last_node = heapFindLastNode(root);
        }
        else{
            new_min->parent->right = NULL;
            root->last_node = new_min->parent->left;
        }
        new_min->left = min_node->left;
        if(new_min->left)
            new_min->left->parent = new_min;
        new_min->right = min_node->right;
        if (new_min->right)
            new_min->right->parent = new_min;
        new_min->parent = NULL;
        root->root_node = new_min;
        heapIncreseKey(root, new_min, new_min->key);
    }
    heapInitNode(min_node);
    return min_node;
}
```

Quadro 22 – Procedimento para remoção do nó com a maior prioridade no heap binário
Fonte: Autoria própria

APÊNDICE C - Procedimentos utilizados pela árvore Van Emde Boas

Possuindo como parâmetro um nó VEB, a chave a ser inserida, a lista de vértices relacionados à chave e o número de chaves distintas aceitas pelo intervalo do nó, o procedimento veb_tree_add, ilustrado no Quadro 23, gerencia a inserção de um nó na estrutura VEB. As linhas 2 a 14 ilustram o caso base onde não há um nó VEB já alocado para salvar a chave inserida. As linhas 16 a 24 fazem a inserção de uma chave em um nó que já possui uma chave com o mesmo valor. As linhas 16 a 21 realizam o tratamento para o caso em que a chave a ser inserida possui o mesmo valor do campo *min* e as linhas 22 e 23 realizam o procedimento análogo para o campo *max*. Com a inserção de uma chave repetida, o vértice associado à chave deve ser incluído na lista de vértices já existente no nó, processo este realizado em tempo constante pelo procedimento auxiliar *merge* que atua nas listas envolvidas.

```
1  vEB* vEB_tree_add(vEB *veb, int x, listNode *list, int u){
2      if(!veb){
3          veb = (vEB*)malloc(sizeof(vEB));
4          veb->min = veb->max = x;
5          veb->u = u;
6          veb->listMin = veb->listMax = list;
7          veb->summary = NULL;
8          if(u > 2){
9              veb->cluster = (vEB**)calloc(sqrt(u),sizeof(vEB*));
10         }
11         else{
12             veb->cluster = NULL;
13         }
14         return veb;
15     }
16     else if(veb->min == x){
17         merge(&(veb->listMin), &list);
18         if(veb->max == x){
19             veb->listMax = veb->listMin;
20         }
21     }
22     else if(veb->max == x){
23         merge(&(veb->listMax), &list);
24     }
25     else{
26         if(x < veb->min){
27             if(veb->min == veb->max){
28                 veb->min = x;
29                 veb->listMin = list;
30                 return veb;
```

```
31              }
32              int aux = veb->min;
33              veb->min = x;
34              x = aux;
35              listNode *listAux = veb->listMin;
36              veb->listMin = list;
37              list = listAux;
38          }
39          else if(x > veb->max){
40              int aux = veb->max;
41              veb->max = x;
42              x = aux;
43              listNode *listAux = veb->listMax;
44              veb->listMax = list;
45              list = listAux;
46              if(veb->min == x)
47                  return veb;
48          }
49          if(u > 2){
50              int high_value = high(x, u), square_of_u = sqrt(u);
51              if(veb->cluster[high_value] == NULL){
52                  listNode *node = initNode(1);
53                  veb->summary = vEB_tree_add(veb->summary,
    high_value, node, square_of_u);
54              }
55              veb->cluster[high_value] =
    vEB_tree_add(veb->cluster[high_value], low(x, u), list,
    square_of_u);
56          }
57      }
58      return veb;
59  }
```

Quadro 23 – Procedimento para adição de um nó em uma Van Emde Boas
Fonte: Autoria própria

A inserção de um valor não existente na árvore pode resultar na atualização do intervalo de valores do nó que acomoda a chave, seja ele o nó recebido como parâmetro ou um de seus filhos localizados no campo *cluster*. Caso a chave inserida seja menor que o limite inferior do nó atual, as linhas 27 a 37 atualizam os campos necessários. Caso o nó possua apenas uma chave (linha 27), basta ao procedimento expandir o intervalo de valores do nó (linhas 28 e 29) encerrando sua chamada. Caso contrário a chave inserida assume o campo *min* e o antigo valor salvo no campo é

enviado para o campo *cluster* na próxima chamada recursiva. Analogamente, as linhas 39 a 47 executam o processo para o campo *max*.

Caso uma chave (seja a chave recebida como parâmetro ou anteriormente salva nos campos *min* ou *max*) deva ser enviada a um subconjunto referenciado pelo *cluster*, uma nova chamada recursiva será efetuada na linha 55. O valor da chave é ajustado ao novo intervalo (square_of_u) com o auxílio do procedimento high() e caso o subconjunto não tenha sido criado anteriormente, sua alocação será feita na chamada da linha 55 e seu índice inserido no *resumo* do nó na linha 53.

O procedimento para remoção de uma chave na VEB, ilustrado no Quadro 24, tem como parâmetro o nó VEB para verificação (inicialmente o nó raiz), a chave a ser removida, o vértice associado à chave e o número de itens coberto pelo intervalo de valores do nó. O procedimento assume que o elemento a ser removido encontra-se na estrutura.

O procedimento, assim como o de inserção, gerencia a memória utilizada pela árvore. O procedimento de remoção é responsável pela liberação do espaço de memória utilizado por um nó alocado pelo procedimento veb_add. Este processo ocorre apenas se a chave a ser removida está presente no nó (linhas 2 a 19). Caso haja apenas uma chave no nó e mais de um vértice associado a ela a exclusão do nó não é permitida, deve-se apenas remover o vértice associado à chave da lista de vértices do nó.

Caso o nó em questão possua um intervalo de valores que comporta apenas duas chaves, a remoção de um item é incondicionalmente feita em uma delas (linhas 20 a 39). A remoção de um vértice da lista de itens associados à uma chave é executada em tempo constante com auxílio do procedimento removeByKey(). Caso o vértice removido seja o único associado à chave requerida, o intervalo de valores do nó é ajustado (linhas 25 a 27 e 34 a 36).

Caso a chave a ser removida esteja localizada no campo *min* (linhas 41 a 63), o vértice desejado é removido da lista referenciada por *listMin*. Se houver mais vértices associados à chave, o procedimento encerra a chamada recursiva. Caso contrário uma nova chave deve ser direcionada para o campo *min*. O novo valor do campo deve ser a menor chave contida nos subconjuntos do nó. A chave em questão é removida e sua lista de vértices associada ao campo *listMin*. Caso o subconjunto se torne vazio, seu índice é removido do conjunto *resumo*. Analogamente o processo é repetido para o campo *max* (linhas 64 a 86), preenchendo-o com a maior chave do *cluster*.

```
1 vEB* vEB_tree_delete(vEB *veb, int x, listNode *vertexNode, int
    u){
2     if(veb->min == veb->max){
3         if(veb->listMin == NULL || veb->listMax == NULL){
```

```
4               if(veb->cluster)
5                   free(veb->cluster);
6               free(veb);
7               return NULL;
8           }
9           removeByKey(&(veb->listMin), vertexNode);
10          if(veb->listMin != NULL){
11              veb->listMax = veb->listMin;
12          }
13          else{
14              if(veb->cluster)
15                  free(veb->cluster);
16              free(veb);
17              return NULL;
18          }
19      }
20      else if(u == 2){
21          if(x == 0){
22              if(veb->listMin != NULL){
23                  removeByKey(&(veb->listMin), vertexNode);
24              }
25              if(veb->listMin == NULL){
26                  veb->min = 1;
27                  veb->listMin = veb->listMax;
28              }
29          }
30          else{
31              if(veb->listMax != NULL){
32                  removeByKey(&(veb->listMax), vertexNode);
33              }
34              if(veb->listMax == NULL){
35                  veb->max = 0;
36                  veb->listMax = veb->listMin;
37              }
38          }
39      }
40      else{
41          if(x == veb->min){
42              if(veb->listMin != NULL){
43                  removeByKey(&(veb->listMin), vertexNode);
44              }
45              if(veb->listMin == NULL){
```

```
46                  if(veb->summary){
47                      int square_value = sqrt(u);
48                      int first_cluster =
    vEB_tree_Minimum(veb->summary);
49                      int new_min =
    vEB_tree_Minimum(veb->cluster[first_cluster]);
50                      veb->min = first_cluster * square_value +
    new_min;
51                      veb->listMin =
    (veb->cluster[first_cluster])->listMin;
52                      (veb->cluster[first_cluster])->listMin = NULL;
53                      veb->cluster[first_cluster] =
54
    vEB_tree_delete(veb->cluster[first_cluster], new_min,
    vertexNode, square_value);
55                      if(veb->cluster[first_cluster] == NULL)
56                          veb->summary =
    vEB_tree_delete(veb->summary, first_cluster, NULL,
    square_value);
57                  }
58                  else{
59                      veb->min = veb->max;
60                      veb->listMin = veb->listMax;
61                  }
62              }
63          }
64          else if(x == veb->max){
65              if(veb->listMax != NULL){
66                  removeByKey(&(veb->listMax), vertexNode);
67              }
68              if(veb->listMax == NULL){
69                  if(veb->summary){
70                      int square_value = sqrt(u);
71                      int last_cluster =
    vEB_tree_Maximum(veb->summary);
72                      int new_max =
    vEB_tree_Maximum(veb->cluster[last_cluster]);
73                      veb->max = last_cluster * square_value +
    new_max;
74                      veb->listMax =
    veb->cluster[last_cluster]->listMax;
75                      veb->cluster[last_cluster]->listMax = NULL;
```

```
76                      veb->cluster[last_cluster] =
77
    vEB_tree_delete(veb->cluster[last_cluster], new_max,
    vertexNode, square_value);
78                      if(veb->cluster[last_cluster] == NULL)
79                          veb->summary =
    vEB_tree_delete(veb->summary, last_cluster, NULL,
    square_value);
80                  }
81                  else{
82                      veb->max = veb->min;
83                      veb->listMax = veb->listMin;
84                  }
85              }
86          }
87          else{
88              int high_value = high(x, u);
89              int square_value = sqrt(u);
90              veb->cluster[high_value] =
91                  vEB_tree_delete(veb->cluster[high_value], low(x,
    u), vertexNode, square_value);
92              if(veb->cluster[high_value] == NULL)
93                  veb->summary = vEB_tree_delete(veb->summary,
    high_value, NULL, square_value);
94          }
95      }
96      return veb;
97  }
```

Quadro 24 – Procedimento para remoção de um nó em uma Van Emde Boas
Fonte: Autoria própria

Há ainda a situação onde a chave pesquisada não se encontra nos campos *min* e *max*, mas sim em um subconjunto referenciado por *cluster*. Neste caso, o procedimento executa uma chamada recursiva a partir do subconjunto calculado até que a chave se enquadre em um dos casos descritos anteriormente. Caso a remoção da chave no subconjunto em questão o torne vazio, seu índice também será removido do campo *resumo*.

O procedimento para atualização de chave, como visto no Quadro 11, referente às equivalencias de operações em uma VEB, é resultado da remoção do item com a prioriodade antiga e a sua reinserção na árvore com o valor atualizado. O procedimento VEB_tree_decrease_key, ilustrado no Quadro 25, recebe como parâmetro

o vértice relacionado, seu antigo custo e o novo valor a ser associado a ele na fila de prioridade. Basicamente o procedimento executa chamadas às funções de remoção e inserção de chaves.

```
1  int vEB_tree_decrease_key(vEB **veb, int old_cost, int new_cost,
       listNode *vertexNode, int u){
2
3      if(!*veb)
4          return 1;
5      *veb = vEB_tree_delete(*veb, old_cost, vertexNode, u);
6      vertexNode->next = vertexNode;
7      vertexNode->prev = vertexNode;
8      *veb = vEB_tree_insert(*veb, new_cost, vertexNode, u);
9      return 0;
10 }
```

Quadro 25 – Procedimento para aumento de prioridade de um nó em uma VEB
Fonte: Autoria própria

A extração do item de maior prioridade na estrutura, analogamente ao procedimento decrementaKey, também é uma combinação de funções suportadas pela VEB. O procedimento veb_tree_extract_min, ilustrado no Quadro 26, executa a remoção de um elemento com a menor chave, após a identificação do mesmo.

```
1  vertex *vEB_tree_extract_min(vEB **veb, int u){
2      if(!*veb)
3          return NULL;
4      int min_value = vEB_tree_Minimum(*veb);
5      vertex *min_vertex = vEB_tree_Minimum_vertex(*veb);
6      *veb = vEB_tree_delete(*veb, min_value, (*veb)->listMin, u);
7      return min_vertex;
8  }
```

Quadro 26 – Procedimento para remoção do nó com maior prioridade na VEB
Fonte: Autoria própria