Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma análise dos algoritmos do simulador ONS

Rafael Lourenço de Lima Chehab

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. André Costa Drummond

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma análise dos algoritmos do simulador ONS

Rafael Lourenço de Lima Chehab

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. André Costa Drummond (Orientador)
CIC/UnB

Prof. Dr. Guilherme Novaes Ramos
CIC / UnB

Prof. Dr. João José Costa Gondim
CIC / UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 7 de julho de 2019

Dedicatória

Dedico esta tese ao meu avô Professor Lourenço Nassib Chehab (*in memoriam*) e a meu pai Mauro Carvalho Chehab, pelos exemplos de integridade e busca constante de conhecimento.

Agradecimentos

Agradeço à Deus, por ter me concedido saúde e força.

Agradeço aos meus pais, irmãos e familiares pela estrutura, dedicação e carinho.

Agradeço ao meu orientador o Professor Dr. André Costa Drummond pela dedicação e orientação. Obrigado por esclarecer minhas dúvidas e ser atencioso e paciente.

Agradeço ao grupo GET (Grupo de Pesquisa em Engenharia de Tráfego em Redes Ópticas), em especial a Lucas Rodrigues Costa, pelo apoio no entendimento e uso do simulador ONS.

Agradeço a todos os professores que contribuíram para a minha trajetória acadêmica desde a educação fundamental até a universidade.

Agradeço aos professores Dr. Guilherme Novaes Ramos e Dr. Vinicius Ruela Pereira Borges pela oportunidade de participar na experiência da Maratona de Programação e da ICPC.

Agradeço aos professores Dr. Ricardo L. de Queiroz e Dr.ª Alba Cristina Magalhães Alves de Melo pelas orientações e conversas ao longo da minha formação, que contribuíram na busca do conhecimento através de suas competência, disponibilidade e simpatia.

Aos amigos e colegas que colaboraram e incentivaram minha formação ao longo da jornada acadêmica.

Agradeço por fim à Universidade de Brasília, tanto por seu quadro acadêmico como pelo pessoal técnico administrativo, por proporcionar um ambiente criativo e amigável para meus estudos.

Resumo

Esse trabalho apresenta uma análise dos algoritmos utilizados no simulador ONS (Optical Networks Simulator), com objetivo de proposta de mudanças que melhorem o tempo de execução do simulador.

Para isso, são estudadas as complexidades assintóticas de várias funcionalidades implementadas no simulador e, com base nisso, são propostas novas soluções que melhoram essa complexidade.

Além disso, é executado um *profiling* do código para identificar pontos de gargalo e procuram-se soluções que melhorem seu tempo de execução.

Ao final, é feito um teste do desempenho baseado em usos realísticos do simulador e são decididas quais funções possuem uma melhoria significativa e que, portanto, serão recomendadas para adição ao simulador e quais não serão.

Palavras-chave: complexidade assintótica, algoritmos e estruturas de dados, redes de computador, redes ópticas elásticas

Abstract

This work presents an study of the algorithms of the Optical Networks Simulator (ONS), proposing potential changes to them with the objective of improving the simulation's execution time.

Therefore, the asymptotic complexities of several functions of the simulator are studied and solutions that improve this complexity are proposed.

Additionally, a code profiling is executed in order to identify bottlenecks which are analyzed and improvements to their time complexity are proposed.

To conclude, the proposed solutions are tested using real applications of the simulator and it's analyzed which ideas had a significant improvement and, thus, will be recommended as an addition to the simulator and which ones won't.

Keywords: asymptotic complexity, algorithms and data structures, computer networks, elastic optical networks

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

EON Elastic Optical Networks.

mAdap m Adaptive RSA algorithms.

OFDM Orthogonal Frequency Division Multiplexing.

ONS Optical Networks Simulator.

RMLSA Routing, Modulation Level and Spectrum Allocation.

RSA Routing and Spectrum Assignment.

SSSP Single-Source Shortest Path.

WDM Wavelength-division Multiplexing.

Capítulo 1

Introdução

1.1 Motivação

As redes ópticas elásticas EON (do inglês Elastic Optical Networks) têm sido estudadas na academia devido ao seu potencial de gerar um serviço flexível e escalável às crescentes demandas de tráfego na Internet [2], [3], [4], [5], [6].

Para poder avaliar soluções dos problemas relacionados à EON é necessário o uso de simulações, uma vez que não é possível a criação de um modelo matemático devido a sua alta complexidade e também não se consegue implementar em um ambiente real devido ao custo e disponibilidade da tecnologia [7].

Assim, foi criado o simulador Optical Networks Simulator (ONS) [1], [8], um simulador de eventos discretos desenvolvido em Java, que permite a análise de tráfego dinâmico em redes com tecnologia de multiplexação por divisão de comprimento de onda (WDM) e de alocação elástica de espectro (EON).

Para a sua implementação utilizou-se um conjunto de algoritmos computacionais essenciais a análise de redes, que formam a base matemática de toda a simulação. Devido à isso, a escolha e implementação desses algoritmos afeta o tempo de execução da simulação como um todo, podendo gerar execuções bastante demoradas.

Em virtude disso, a motivação desse trabalho é a análise dos algoritmos implementados no ONS, propondo novas soluções ou melhorias aos algoritmos atuais de forma a diminuir o tempo de execução das simulações.

1.2 Problema

Na literatura, encontram-se vários problemas de computação que possuem diversas soluções. Sobre o ponto de vista de tempo de execução, essas soluções podem ser

comparadas pela sua complexidade assintótica e dada uma implementação pode-se analisar o desempenho em um conjunto de execuções.

A escolha de um algoritmo que possua um desempenho desejado requer um amplo estudo sobre os vários algoritmos existentes e suas aplicações, assim como um conhecimento de técnicas, paradigmas e métodos para a análise de problemas não clássicos.

Assim, serão buscados na literatura várias soluções para problemas clássicos utilizadas no simulador ONS e serão aplicadas técnicas e paradigmas para obter algoritmos para problemas não clássicos, ou seja, problemas específicos do simulador que não possuem uma análise na literatura. Adicionalmente, serão feitas melhorias na implementação de outros algoritmos que, apesar de não alterarem a complexidade assintótica, melhorem o desempenho nas execuções da simulação

1.3 Objetivos

O objetivo principal desse trabalho é obter uma diminuição significativa do tempo de execução total do simulador. Para isso, são definidos os seguintes objetivos específicos:

- Melhoria da complexidade assintótica do ONS, através da aplicação de algoritmos clássicos.
- Realização de um *profiling* do código e proposição de soluções que reduzam o tempo de execução das linhas mais executadas.
- Análise dos resultados das soluções propostas para vários casos de execução, decidindo quais mudanças serão incorporadas ao simulador.

Capítulo 2

Algoritmos e Complexidade

Este capítulo tem como objetivo apresentar os conceitos e algoritmos utilizados na realização deste trabalho. Primeiramente, serão explicados os conceitos de análise assintótica e notação big-O. Em seguida, serão explicadas estruturas de dados, com suas respectivas complexidades assintóticas. Por fim, serão abordados problemas de programação que estão presentes no simulador e serão apresentados os algoritmos e paradigmas de programação que os resolvem.

2.1 Complexidade de Tempo e Análise Assintótica

2.1.1 Algoritmos e Complexidade de Tempo

Em 2009, Cormen et al. [9] definiram algoritmo como uma série de passos computacionais bem definidos que recebem um valor ou conjunto de valores, chamado *input*, e produz um valor ou conjunto de valores, chamado *output*. Esses algoritmos são utilizados para resolver problemas computacionais, os quais utilizam o *input* para passar informações importantes sobre o problema e que utilizam o *output* para obter a resposta desejada. Assim, a computação envolve desenvolver algoritmos que consigam resolver certos problemas propostos.

No entanto, mesmo que haja-se encontrado uma solução para o problema, é possível que, na prática, essa solução seja inviável, uma vez que ela pode requerer muito tempo e/ou memória para seu funcionamento [10], tornando-se necessário uma análise mais profunda do problema e a proposta de uma nova solução que execute mais rapidamente.

2.1.2 Análise Assintótica

Para analisar a complexidade de uma solução, pode-se executá-la e observar o tempo de sua execução e a quantidade de memória total utilizada. Porém, esses parâmetros dependem de diversos fatores, incluindo o processador e tipo de memória utilizados, otimizações de compilação, entre outros. Isso dificulta a comparação entre diversos algoritmos, o que levou a busca por um parâmetro que não dependesse de fatores externos como o hardware.

Dessa forma, surge uma nova forma de analisar-se o algoritmo, através de uma função que depende do programa e cujos parâmetros são a entrada do programa. Essa função indica o quanto o tempo de execução do programa cresce a medida que a entrada cresce [10].

Em particular, estamos interessados na execução para grandes valores de entrada, os quais podem potencialmente demorar mais do que entradas de menor valor, em outras palavras, analisa-se essa função no limite, para entradas ilimitadas, o que denomina-se análise assintótica.

Por exemplo, verificando o seguinte pedaço de código:

```
1: for i ← 0, n − 1 do
2:     for j ← n − 1, i + 1 do
3:         if a_{j−1} < a_j then
4:             tmp ← a_{j−1}
5:             a_{j−1} ← a_j
6:             a_j ← tmp
7:         end if
8:     end for
9: end for
```

Assumindo-se que leva-se tempo t_i para executar a linha i desse algoritmo, o qual é uma constante dependente do hardware no qual o programa foi executado.

Observando a linha 1, tem-se um loop que é executado n vezes, para i variando de 0 a $n-1$.

Já na linha 2, o loop é executado de $n-1$ até $i+1$. Logo, para contabilizar-se a quantidade de vezes em que essa linha é executada deve-se achar o valor de $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-i-1) = n*(n-1) - \sum_{i=0}^{n-1} i = n*(n-1) - n*(n-1)/2 = n*(n-1)/2$.

Perceba que a linha 9 é executada a mesma quantidade de vezes que a linha 1, ou seja n vezes e as linhas 3, 4, 5, 6, 7 e 8 são executadas a mesma quantidade de vezes que a linha 2, ou seja $n*(n-1)/2$ vezes.

Assim a função que mede o crescimento da função em relação à entrada, ou seja, a função utilizada na análise assintótica é $f(n) = n * (t_1 + t_9) + n * (n-1)/2 * \sum_{i=2}^{8} t_i = n^2 * (\sum_{i=2}^{8} t_i/2) + n * (t_1 + t_9 - \sum_{i=2}^{8} t_i/2)$. Porém deseja-se analisar essa função no limite, para n grande. Assim, a potência de maior ordem possui mais relevância no tempo de execução, o que possibilita a simplificação de f para $f(n) = n^2 * (\sum_{i=2}^{8} t_i/2) = C * n^2$ para C uma constante dependente do hardware. O que conclui a análise assintótica desse código.

2.1.3 Notação big O

Dada uma função $g(n)$, denota-se $O(g(n))$ como o conjunto de funções que respeita a seguinte propriedade [11]:

$$O(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{R}_{>0} \mid |f(n)| \leq c * g(n), \forall n \geq n_0\} \tag{2.1}$$

Em outras palavras, para valores muito grandes (superiores a n_0), a função $f(n)$, a qual representa o algoritmo analisado, é limitada **superiormente** pela função $g(n)$.

Para a função $f(n)$ obtida no algoritmo acima, tem-se que $f(n) \in O(n^2)$. Poderia dizer-se também que $f(n) \in O(n^3)$, porém nesse trabalho será buscado o limite superior (notação big Oh) menor possível para o algoritmo analisado.

2.1.4 Notação big Omega

Dada uma função $g(n)$, denota-se $\Omega(g(n))$ como o conjunto de funções que respeita a seguinte propriedade [11]:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{R}_{>0} \mid f(n) \geq c * g(n), \forall n \geq n_0\} \tag{2.2}$$

Em outras palavras, para valores muito grandes (superiores a n_0), a função $f(n)$, a qual representa o algoritmo analisado, é limitada **inferiormente** pela função $g(n)$.

Para a função $f(n)$ obtida no algoritmo acima, tem-se que $f(n) \in \Omega(n^2)$. Poderia dizer-se também que $f(n) \in \Omega(n)$.

2.1.5 Notação big Theta

Dada uma função $g(n)$, denota-se $\Theta(g(n))$ como o conjunto de funções que respeita a seguinte propriedade [11]:

$$\Theta(g(n)) = \{f(n) : \exists c, c', n_0 \in \mathbb{R}_{>0} \mid c * g(n) \leq f(n) \leq c' * g(n), \forall n \geq n_0\} \tag{2.3}$$

Em outras palavras, para valores muito grandes (superiores a n_0), a função $f(n)$, a qual representa o algoritmo analisado, é limitada **inferiormente** e **superiormente** pela função $g(n)$.

Para a função $f(n)$ obtida no algoritmo acima, tem-se que $f(n) \in \Theta(n^2)$, porém $f(n) \notin \Theta(n)$ e $f(n) \notin \Theta(n^3)$.

2.1.6 Análise Amortizada

Existem estruturas de dados sobre as quais são executadas um número n de operações tal que uma única operação pode ter uma complexidade alta, mas o tempo total gasto, ou seja, a soma do tempo de cada operação tenha uma boa complexidade.

Nesses casos, é utilizada a análise amortizada, a qual indica o tempo médio gasto em 1 operação entre n operações executadas [9]. É importante ressaltar que a análise amortizada garante que esse desempenho mesmo no pior caso. Em outras palavras, assumindo que após a análise amortizada de uma estrutura de dados, chegou-se que todas as funções são $O(lg(n))$, isso significa que **garantidamente** o tempo de execução de **todas** as n operações executadas é $O(n * lg(n))$, embora o tempo de execução de **1** operação específica possa ser $O(n)$, por exemplo.

Método Potencial

Um dos métodos utilizados para a análise amortizada, o qual será utilizado posteriormente nesse trabalho, é o método potencial.

Nesse método associa-se um potencial à estrutura de dado como um todo, ao invés de objetos ou funções específicas. Assumindo que foram executadas n operações a partir de um estado inicial D_0 da estrutura de dados. Cada operação pode alterar a estrutura de dados e, logo, denomina-se D_i como o estado da estrutura de dados após a i-ésima operação [9].

Denomina-se a função potencial Φ de um estado D_i para um número real $\Phi(D_i)$. Essa função é utilizada para estimar o tempo gasto para as operações, permitindo que em uma execução seja estimado um valor (seja ele c_i^*) acima do custo real dessa operação (seja ele c_i) e que para outra execução o valor estimado seja menor que o valor real.

Perceba que se

$$\sum_{i=1}^{n} c_i^* \geq \sum_{i=1}^{n} c_i \tag{2.4}$$

, então o tempo total **real** das n operações será $O(\sum_{i=1}^{n} c_i^*)$ (verificar seção 2.1.3) e o tempo amortizado de 1 operação é $O(\sum_{i=1}^{n} c_i^*/n)$. Caso $\sum_{i=1}^{n} c_i^*$ não fosse necessariamente maior

ou igual a $\sum_{i=1}^{n} c_i$, então não poderia-se utilizar essa função potencial para a análise amortizada.

No método potencial, o custo estimado, também chamado custo amortizado é

$$c_i^* = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{2.5}$$

Ou seja, ele é o custo real somado da diferença de potencial entre o estado atual da estrutura de dados e o estado anterior. Perceba que $c_i^* \geq c_i \iff \Phi(D_i) \geq \Phi(D_{i-1})$ e analogamente, $c_i^* < c_i \iff \Phi(D_i) < \Phi(D_{i-1})$.

Com base na Equação 2.5, o tempo total de execução é:

$$\sum_{i=1}^{n} c_i^* = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \Phi(D_n) - \Phi(D_0) + \sum_{i=1}^{n} c_i \tag{2.6}$$

Assim, tem-se que se $\Phi(D_n) \geq \Phi(D_0)$ para cumprir a condição da Equação 2.4 (perceba que para $i > 1$, D_i não precisa ser maior que D_{i-1}, o que permite que $c_i^* < c_i$ para certo valor de i).

Em uma execução genérica, não se conhece o valor de n, logo deve-se ter:

$$\Phi(D_i) \geq \Phi(D_0) \tag{2.7}$$

Para demonstrar a aplicação da função potencial, considera-se a estrutura de dados pilha com uma função adicional. (Exemplo adaptado de [9]).

As operações básicas da pilha são:

S.push(x), a qual coloca o objeto x no final (*bottom*) da pilha e

S.pop(), a qual retira o objeto que está no final (*bottom*) da pilha, caso ele exista, caso contrário não altera nada.

Além disso, será adicionada a operação:

S.multipop(k), a qual executa a operação *S.pop*() k vezes. Caso a pilha tenha menos de k elementos, então o procedimento termina quando a pilha se esvazia.

Implementando a pilha com um vetor, indexado a partir de 0 e um contador n o qual indica a quantidade de elementos na pilha, a operação *pop* pode ser implementado em tempo $O(1)$, obtendo o elemento na posição $n - 1$ (*bottom*) e decrementando n, e a operação *push* pode ser implementada em $O(1)$ colocando o elemento na posição n (novo *bottom*) e incrementando n.

Além disso, a operação adicional *multipop*(k) será executada em tempo $O(min(n, k))$, uma vez que pela definição executa-se *pop* no máximo k vezes, porém se a pilha for esvaziada $n = 0$ o procedimento para. Logo, se $k < n$, o execução é $O(k)$ e se $n < k$ a execução é $O(n)$. Juntando esses casos, tem-se $O(min(n, k))$

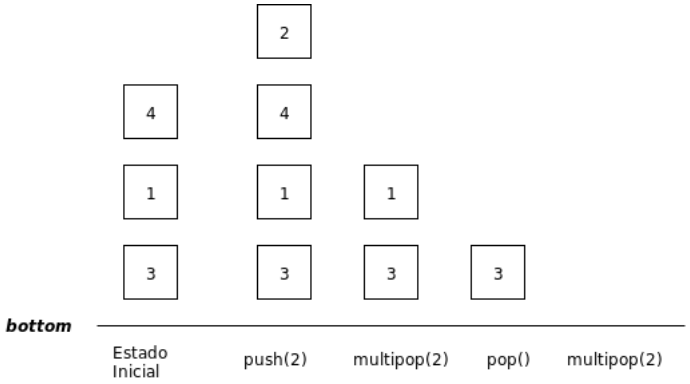

Figura 2.1: Pilha com *multipop*.

Perceba que a análise acima é o tempo real de execução das operações, ou seja c_i.

A função potencial Φ será definida como a quantidade de elementos na pilha, logo $\Phi(D_i) = n_i$. Assumindo que começa-se com uma pilha vazia $\Phi(D_0) = 0$ e como $n \geq 0$ (uma vez que a pilha não pode ter um número negativo de elementos), a Equação 2.7 é cumprida, o que garante que a análise amortizada está correta.

Analisando as operações de acordo com a função potencial, para a função *push*, tem-se:

$\Phi(D_i) - \Phi(D_{i-1}) = n_i - n_{i-1} = (n_{i-1} + 1) - n_{i-1} = 1$

Logo: $c_i^* = c_i + \Phi(D_i) - \Phi(D_{i-1}) = O(1) + 1 = O(1)$.

Para a função *multipop*, tem-se que ao final do prodecimento foram retiradas $k^* = min(n, k)$ elementos da lista, logo:

$\Phi(D_i) - \Phi(D_{i-1}) = -k^*$

Assim, $c_i^* = c_i + \Phi(D_i) - \Phi(D_{i-1}) = O(k^*) - k^* = O(1)$.

Analogamente, para o *pop*, $c_i^* = O(1)$. Assim, todas as função da pilha são amortizadamente $O(1)$ e x operações na pilha rodam em tempo $O(x)$ (**não** amortizadamente).

2.2 Estruturas de Dados

Essa seção explica o funcionamento e implementação de estruturas de dados utilizadas nesse trabalho, analisando suas complexidades assintóticas de tempo e espaço.

2.2.1 Resizable-Array

A primeira estrutura de dados a ser explicada é o Resizable-Array, uma estrutura simples, porém essencial para a redução do tempo de execução de vários algoritmos.

Essa estrutura de dados implementa um array que permite a expansão ou redução de seu tamanho de uma maneira rápida. Especificamente, analogamente a um array comum, as operações de atribuição e referência ao i-ésimo elemento e referência ao tamanho do array funcionam em $O(1)$ e uma inserção ou remoção de um elemento no meio do array é $O(n)$.

Porém, a adição de um novo elemento ao final do array terá complexidade $O(1)$ amortizada em um Resizable-Array [9], [12], comparado com $O(n)$ para um array comum.

A implementação dessa estrutura de dados se dá através de uma alocação de mais elementos do que o pedido, ou seja, em uma Resizable-Array, tem-se o tamanho pedido, seja ele n, e o tamanho real, seja ele $size$. Assim, a adição de um novo elemento no final da fila se dá com o seguinte código:

```
1:  int n, size
2:  Element[] array
3:  procedure ADD(Element e)
4:      if n < size then
5:          array_n ← e
6:          n ← n + 1
7:      else
8:          new_array = new Element[2 * size]
9:          for i ← 0, size − 1 do
10:             new_array_i ← array_i
11:         end for
12:         array ← new_array
13:         size ← 2 * size
14:         array_n ← e
15:         n ← n + 1
16:     end if
17: end procedure
```

Analisando a execução dessa função, e assumindo que inicialmente $n = size$. Na primeira chamada desse programa, na linha 4 a condição será falsa, será executado o else da linha 7, o qual executa n atribuições e logo possui complexidade O(n).

Em seguida, teremos $size_{new} = 2 * size$, ou seja, $size_{new} = n + n$ e $n_{new} = n + 1$, assim, $n_{new} < size_{new}$. Logo, nas próximas chamadas, a condição da linha 4 será verdadeira e o código do if será executado, o qual possui somente 1 atribuição e 1 incremento e logo é O(1). Perceba-se que a linha 4 será verdadeira enquanto $n_{new} < size_{new}$ e em cada chamada n_{new} é incrementado de 1. Assim, esse loop será chamado $size_{new} - n_{new} = n - 1$

vezes.

Assim, de acordo com os parágrafos acima, haverá 1 execução em tempo O(n) e $n - 1$ execuções O(1), o que implica que n execuções levarão o tempo $O(1 * n + (n - 1) * 1) = O(2 * n - 1) = O(n)$. Assim, 1 execução individual é amortizadamente $O(n)/n = O(n/n) = O(1)$.

Essa estrutura de dados já está implementada em Java na classe ArrayList [13].

2.2.2 Binary Search Tree

Search Trees são estruturas de dados que permitem varias operações dinâmicas, como busca, mínimo, máximo, inserção e deleção [9], que permitem seu uso como um dicionário de busca ou como uma fila de prioridades.

Essa estrutura de dados pode ser modelada como uma árvore binária, recebendo o nome *Binary Search Tree*, primairamente definida em [14], onde cada elemento é chamado de nó e cada nó pode ter até dois nós como filhos, chamados filho da esquerda e filho da direita. Adicionalmente, garante-se as seguintes propriedades:

Assumindo um nó cujo elemento tenha valor x.

- O filho da esquerda e todos os nós de sua subárvore possuem valor <= x.
- O filho da direita e todos os nós de sua subárvore possuem valor > x.

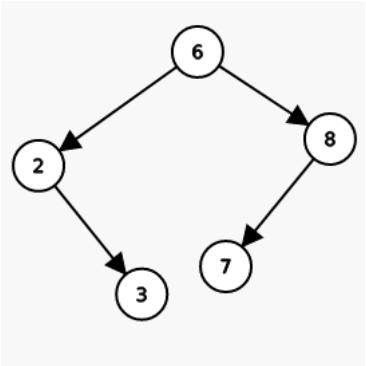

Figura 2.2: Binary Search Tree.

Devido à isso, a busca por um elemento na árvore é feita de maneira similar à uma busca binária, se a raiz possui valor igual ao que se busca, a busca termina. Se o elemento buscado é maior que a raiz busca na sub-árvore da direita. Caso contrário, busca-se na sub-árvore da esquerda. As operações de mínimo e máximo também funcionam através de uma decida na árvore

Perceba-se que no pior caso o tempo de execução desses algoritmos será proporcional a altura da árvore. No pior caso, assumindo-se n elementos na árvore, cada nó poderia ter somente um filho, o que levaria à altura ser n e o tempo de execução seria $O(n)$.

Para evitar isso, modifica-se as operações de inserção e deleção para garantir que a árvore possua altura $O(log(n))$. Há várias estruturas de dados que garantem isso através de diferentes métodos, a elas dá-se o nome coletivo de Binary Search Trees.

Essa estrutura de dados já está implementada em Java na classe TreeSet, a qual utiliza Red-Black Tree.

2.2.3 Heap

Considere o seguinte problema: deseja-se poder adicionar dinamicamente elementos que possuam um valor, que será chamado v_a, podendo ou não incluir outras informações a um conjunto inicialmente vazio e deseja-se poder a qualquer instante de tempo verificar o menor valor e removê-lo.

Esse problema é um problema clássico de programação, que é utilizado em diversas aplicações, como na algoritmo de Dijkstra.

Pode-se pensar um uma solução *naive* que consiste em adicionar o elemento em um vetor e na busca pelo menor elemento passar por todos os elementos do vetor. Essa ideia possui complexidade O(n) para busca, inserção e remoção, para n o tamanho do vetor e utilizando um vetor comum.

Em muitos casos, essa solução não possui um desempenho desejado. Por isso, surgiram algoritmos cujo objetivo é reduzir a complexidade para O(log n).

2.2.4 Binary Min Heap

Como deseja-se obter sempre o menor elemento, faz sentido ordenar os elementos do conjunto. Porém, como os elementos estão constantemente sendo adicionados, seria necessário reordenar o conjunto várias vezes, o que é um processo custoso. Para resolver isso, pode-se utilizar não uma ordenação completa dos elementos, mas uma ordenação parcial.

Nesse contexto, um Binary Min Heap [15] é uma estrutura de dados representada como uma árvore binária completa, na qual define-se $p(x)$, para x um elemento da árvore, como o seu pai (para a raiz $p(x)$ é nula) e $l(x)$ como o filho esquerdo de x e $r(x)$ como o filho direito. Para que essa árvore seja um Min Heap, ela deve seguir a seguinte propriedade:

$$v_a <= v_{l(a)} \&\& v_a <= v_{r(a)} \tag{2.8}$$

, na qual v_x representa o valor do elemento x da árvore

Ou seja, se os elementos dessa árvore fossem ordenados, o elemento a virá depois de o seu pai, seja ele $p(a)$, devido a propriedade acima. Essa propriedade também aplica para o pai $p(a)$ que virá depois de $p(p(a))$. Assim, sabe-se que o elemento a é maior que todos os seus antecessores.

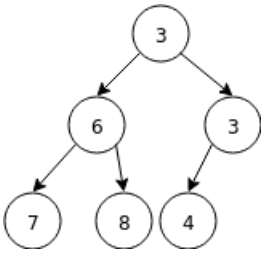

Figura 2.3: Binary Min Heap.

Similarmente, o elemento a será menor que todos os elementos na sua sub-árvore, uma vez que para todo elemento x de sua sub-árvore, a é seu antecessor e logo $v_a <= v_x$.

Devido à essa propriedade sabe-se que a raiz da árvore contêm o menor valor. Deve-se notar porém que para dois elementos a e b tal que a não é antecessor de b e b não é antecessor de a não pode-se dizer qual é menor e qual é maior.

Logo, a operação de obter o menor elemento consiste em retornar a raiz da árvore e logo é O(1).

Para as operações de inserção de um elemento e remoção do mínimo, deve-se analisar melhor o algoritmo. Para começar, explica-se como a árvore do Binary Min Heap pode ser implementada em um vetor.

Como a árvore é completa, ela pode ser implementada como um vetor, onde cada posição i é um elemento e o filho esquerdo de i, $l(i)$, está na posição $2 * i + 1$ e o seu filho direito, $r(i)$ está em $2 * i + 2$, assumindo um vetor iniciado na posição 0. Como pode ser visto na figura Figura 2.4, que representa a mesma árvore da figura Figura 2.3.

Assim, a operação de inserção pode ser feita através da inserção de um elemento no final do vetor e uma operação de atualização nessa posição, a qual irá garantir a propriedade da heap, enquanto a operação de remoção do mínimo pode ser feita através da substituição da raiz (primeiro elemento do vetor) pelo último elemento do vetor e a utilização da mesma operação de atualização na raiz, que contêm o elemento que estava ao final do vetor. Isso implica que a inserção e a remoção tem a mesma complexidade, a qual é a complexidade da operação de atualização.

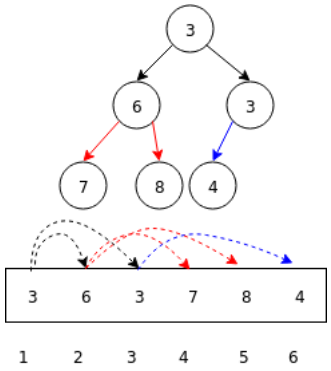

Figura 2.4: Binary Min Heap - Implementação em vetor.

Portanto, para finalizar o estudo do Binary Min Heap, deve-se somente definir a operação de atualização. Seja ela uma função que recebe o elemento do vetor que foi modificado e que, logo pode não seguir a propriedade da Equação 2.8 da heap. Esteja esse elemento na posição i, existem 2 casos:

1. $v_{p(i)} > v_i$, ou seja, a propriedade não está sendo cumprida para o pai. Perceba que como a raiz não possui pai, essa condição vai ocorrer somente na inserção de um elemento (e não na remoção).

   Nesse caso, troca-se os elementos nas posições $p(i)$ e i de forma que $v_{p(i)} <= v_i$.

   Em seguida, chama-se recursivamente a função de atualização para $p(i)$, a qual pode não seguir mais a propriedade da Equação 2.8 até que $v_{p(i)} <= v_i$ ou que tenha-se chegado à raiz.

   Perceba que como o novo valor de $v_p(i)$ é menor que o seu valor anterior, a propriedade da Equação 2.8 continua valendo para i.

   Logo, a chamada recursiva em $p(i)$ sempre subirá um nível na árvore sem nunca descer.

2. $v_i > v_{l(i)}||||v_i > v_{r(i)}$, ou seja, a propriedade não está sendo cumprida para i, com um ou os dois de seus filhos. Perceba que como o último elemento não possui filho, essa condição vai ocorrer somente na remoção.

   Nesse caso, troca-se i com o seu filho de menor valor, digamos que ele esteja na posição x, de forma que: $v_i <= v_{l(i)} \&\& v_i <= v_{r(i)}$.

   Em seguida, chama-se recursivamente a função de atualização para x, o qual pode não seguir mais a propriedade da Equação 2.8 até que $v_x <= v_{l(x)} \&\& v_x <= v_{r(x)}$ ou que não tenha-se mais filhos.

Perceba que devido a propriedade da heap, todos os elementos da sub-árvore de x possuem v maiores ou iguais a $v(x)$, que agora é v_i (tal que $p(x) = i$), logo qualquer mudança feita na sub-árvore de x não afetará $p(x) = i$.

Logo, a chamada recursiva em x sempre descerá um nível na árvore sem nunca subir.

Sabe-se que em uma árvore completa como um Binary Min Heap, a altura da árvore é O(log n). Como a função de atualização ou somente subirá a árvore ou somente descerá, essa função possui complexidade de tempo igual a altura da árvore e logo ela é O(log n).

Isso implica que as operações de inserção e remoção possuem complexidade O(log n).

2.2.5 Fibonacci Heap

A Fibonacci Heap é uma outra possível implementação de uma heap, porém as suas funções rodam em tempo amortizado, sendo necessário o uso do método visto na Seção 2.1.6 para a análise da sua complexidade.

Similar à Binary Min Heap, a Fibonacci Heap [16] também é representada como uma árvore que segue a propriedade da Equação 2.8, na qual o elemento do pai é menor do que o elemento dos filhos, mas na Fibonacci Heap o número de filhos não é limitado a somente 2 (mas dado um número n de elementos na árvore, esse valor possui um limite superior $d(n)$) e a árvore não tem que ser completa. Além disso, a Fibonacci Heap permite a existência de várias árvores, cujas raízes são postas em uma lista duplamente encadeada.

Cada nó da Heap possui sua chave (*key*) e seu valor (*value*), seu pai (*parent*), o ponteiro para um de seus filhos (*child*) e ponteiros $left$ e *right*. Para a raiz da árvore, $left$ e *right* são utilizados para a fila duplamente encadeada das raízes de todas as árvores, a qual é chamada root list e *parent* é nulo. Para qualquer outro nó, $left$ e *right* são utilizados para uma lista duplamente encadeada de filhos do mesmo pai.

Além disso, cada nó possui um valor *degree*, o qual indica o número de filhos que ele possui. Esse valor é importante para a prova da complexidade da Heap e é utilizada na remoção do menor elemento.

A Fibonacci Heap também guarda um ponteiro (*min*) para o menor elemento do Heap e o número n de elementos da heap.

Como mencionado acima, a complexidade será analisada amortizadamente e para isso será utilizada a função potencial. Seja:

$$\Phi(H) = t_i \tag{2.9}$$

Onde t_i é a quantidade de árvores na root list.

Assumindo que a Heap começa vazia, tem-se que $\Phi(D_0) = 0$ e como $\Phi(D_i) \geq 0$, já que $t_i \geq 0$ (a árvore não pode ter um número negativo de nós na root list), a Equação 2.7 é cumprida e logo, o método potencial pode ser usado.

Seja D_{i-1} o estado da Fibonacci Heap em um certo ponto da execução, serão analisadas as operações da heap, assumindo que elas sejam executadas após atingir-se esse ponto da execução do programa, após o qual o estado da Heap será D_i.

Inserção de Elemento

Na inserção de um elemento na heap, cria-se uma nova árvore na qual o novo elemento é a raiz e o único nó da árvore. Logo essa árvore é adicionada na root list e n é acrescido de 1. Assim $\Phi(D_i) = 1 + \Phi(D_{i-1})$ e o custo amortizado c_i^* será $c_i^* = \Phi(D_i) - \Phi(D_{i-1}) + c_i = 1 + O(1) = O(1)$.

Logo essa função possui complexidade amortizada $O(1)$.

Obtenção do Elemento Mínimo

Na obtenção do elemento mínimo, deve-se retornar os valores *key* e *value* do nó *min* que está guardado na Heap. Além disso, $\Phi(D_i) = \Phi(D_{i-1})$, já que a quantidade de elementos da heap não é alterado.

Logo, essa função possui complexidade amortizada $c_i^* = c_i = O(1)$

Remoção do Elemento Mínimo

A remoção do elemento mínimo da Heap pode ser dividido em 3 etapas:

Inicialmente, retira-se os nós filhos de *min* da sua lista de filhos e adiciona-os na root list da Heap, tal como mostra a Figura 2.5. Após o qual o nó mínimo da Heap é deletado.

Em seguida, une-se as árvores com mesma quantidade de filhos em uma árvore só, tal que a árvore cuja raiz possui o menor *value* é o pai da árvore cuja raiz possui o maior *value* (Figura 2.6). Esse procedimento é executado até que não haja 2 árvores com mesmo número de filhos de acordo com o seguinte procedimento:

1: Seja A um array indexado a partir de 0 e com $d(n) + 1$ elementos (range de 0 a $d(n)$).
2: **for** $\forall w \in$ root list **do**
3: $\quad x \leftarrow w$

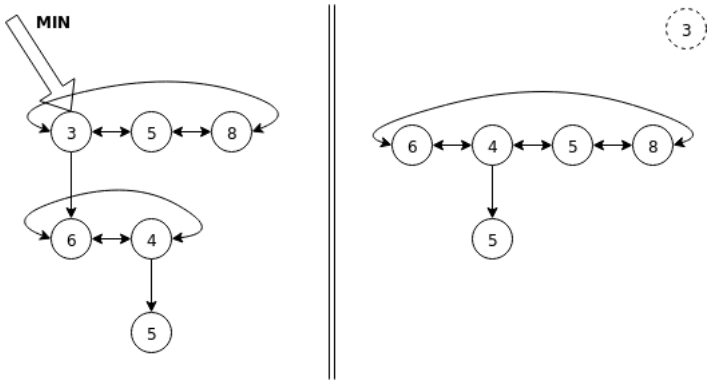

Figura 2.5: Fibonacci Heap - Remoção - Etapa 1.

4: $deg \leftarrow x.degree$
5: **while** $A[deg] \neq null$ **do**
6: $y \leftarrow A[deg]$
7: $A[deg] \leftarrow null$
8: $x \leftarrow merge(x, y)$ ▷ Adiciona o elemento com maior valor entre x e y na lista de filhos do elemento de menor valor entre x e y, o que aumenta o *degree* desse elemento.
9: $deg \leftarrow deg + 1$
10: **end while**
11: $A[deg] \leftarrow x$
12: **end for**

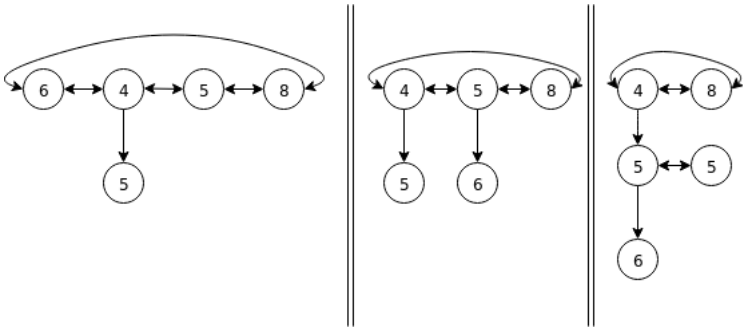

Figura 2.6: Fibonacci Heap - Remoção - Etapa 2.

Ao final, passa-se por todas as árvores na root list, para encontrar a árvore com o menor *value*, a qual será o elemento com o menor *value* de toda a Heap, visto que filhos sempre possuem values maiores ou iguais aos seus pais. A raiz da árvore achada é o novo *min*.

Para calcular a complexidade dessa operação inicialmente será calculado o custo real c_i. A etapa 1 dessa operação possui complexidade $O(d(n))$, onde $d(n)$ indica a maior quantidade de filhos que qualquer nó pode ter em uma Heap de n elementos. Pode ser visto que, como na etapa 2 fez-se com que todas as árvores tivessem número de filhos diferente e $d(n)$ é o limite superior de filhos na Heap, então a etapa 3 é executada em tempo $O(d(n))$.

Falta calcular então o custo da etapa 2, perceba que ao final da etapa 1 (2.2.5), foram adicionados no máximo $d(n)$ elementos à root list (os filhos de *min*) e foi retirado 1 elemento (*min*). Então tem-se $t_{i-1} + d(n) - 1$ na root list ao começo da etapa 2 e o for da linha 2 é executada em $O(t_{i-1} + d(n))$. Pode-se ver que o while da linha 5 também será executado no mesmo tempo, uma vez que devido à função *merge*, sempre que o while for executado a root list terá um elemento a menos e quando houver 1 só elemento, o while não é mais executado. Assim, a etapa 2 possui custo real $O(t_{i-1} + d(n))$. Logo:

$c_i = O(d(n)) + O(d(n) + t_{i-1}) + O(d(n)) = O(d(n) + t_{i-1})$

Em relação à função potencial, tem-se $\Phi(D_{i-1} = t_{i-1}$ pela definição e no máximo $\Phi(D_i) = 1 + d(n)$, devido a que, após a etapa 2, o número de elementos na root list é a quantidade de elementos no array A, com capacidade $1 + d(n)$. Logo:

$c_i^* = O(d(n) + t_{i-1}) + 1 + d(n) - t_{i-1} = O(2 * d(n) - 1) = O(d(n))$

Logo, essa função possui complexidade amortizada de $O(d(n))$, onde $d(n)$ é a maior quantidade de filhos que qualquer nó pode ter em uma Heap de n elementos.

Para terminar a discussão sobre Fibonacci Heap, prova-se que $d(n) = O(lg(n))$.

Seja x um nó na Heap (não necessariamente na root list), com *degree* igual a d. Sejam $y_1, y_2, ..., y_d$ os filhos de x, na ordem em que eles foram adicionados na lista de filhos. Prova-se que $y_i.degree = i - 1$, uma vez que quando y_i for adicionado na lista de filhos, x tinha $i - 1$ filhos $(y_1, ..., y_{i-1})$.

Vale observar que é possível adicionar uma operação de remoção de elementos na Fibonacci Heap, o que alteraria essa prova, porém nesse trabalho, não foi utilizada essa operação.

Em seguida, prova-se que

$$size(x) \geq F_{d+2} \tag{2.10}$$

por indução na altura da sub-árvore de x, na qual define-se altura como o maior caminho entre x e um de seus descendentes, onde *size* indica o número de elementos na árvore de x, incluindo ele mesmo, e F_i é o i-ésimo elemento de Fibonacci, ou seja, F_i segue a seguinte fórmula:

$F_0 = 0$

$F_1 = 1$

$F_k = F_{k-1} + F_{k-2}, \forall k \geq 2.$

Base de indução: Altura de $x = 0$, então $d = 0$ e $size(x) = 1 = F_2 \Rightarrow size(x) \geq F_{d+2}$

Passo Indutivo: Como a altura dos filhos de x, $y_1, ..., y_d$ é estritamente menor que a altura de x, então, por indução, $size(y_i) \geq F_{y_i.degree+2} = F_{i+1}$. Logo:

$size(x) \geq 1 + \sum_{i=1}^{d} F_{i+1} = 1 + \sum_{i=2}^{d+1} F_i \geq 1 + \sum_{i=1}^{d} F_i = 1 + \sum_{i=0}^{d} F_i$, uma vez que $F_{d+1} > F_1$, já que $d > 0 \Rightarrow d + 1 > 1$.

Reescrevendo:

$$size(x) \geq 1 + \sum_{i=0}^{d} F_i \tag{2.11}$$

Em seguida, prova-se que $1 + \sum_{i=0}^{d} F_i = F_{d+2}$. Por indução em d, tem-se:

Base de indução: $d = 0$, então $1 + \sum_{i=0}^{0} F_i = 1 = F_2$

Passo Indutivo: Por indução em d,

$1 + \sum_{i=0}^{d-1} F_i = F_{d+1} \Rightarrow \sum_{i=0}^{d} F_i = F_d + F_{d+1} = F_{d+2}$.

O que conclui a prova de $1 + \sum_{i=0}^{0} F_i = 1 = F_2$.

Logo a Equação 2.11, torna-se $size(x) \geq F_{d+2}$, e a Equação 2.10 foi provada por indução.

Por último, prova-se que $F_{d+2} \geq \phi^d$ por indução no valor de d, onde *phi* é a proporção áurea, ou seja, $phi = (1 + \sqrt{5})/2 = 1.61803$.

Base de indução: Para $d = 0$, então $F_2 = 1 = \phi^0$ e para $d = 1$, $F_3 = 2 > 1.619 > \phi^1$

Para o passo indutivo, utiliza-se o fato de que ϕ é a raiz positiva de $x + 1 = x^2$, ou seja, $\phi + 1 = \phi^2$

Passo Indutivo: Por indução em d, $F_{d-1} \geq \phi^{d-3}$ e $F_{d-2} \geq \phi^{d-4}$. Assim,

$$F_d = F_{d-1} + F_{d-2} \geq \phi^{d-3} + \phi^{d-4} = \phi^{d-4} * (\phi + 1) = \phi^{d-4} * \phi^2 = \phi^{d-2} \tag{2.12}$$

Com isso, a indução está provada e a Equação 2.10 implica em:

$size(x) \geq \phi^d \Rightarrow d \leq lg(size(x)) \Rightarrow d \leq lg(n)$, já que $size(x) \leq n$.

Logo, a função de remoção de mínimo possui complexidade amortizada de $O(lg(n))$.

2.2.6 Multiplicação de Matrizes

A multiplicação de matrizes pode ser matematicamente definido como:

Sejam A, B e C matrizes reais n x n, tais que $A_{i,j}$, $B_{i,j}$ e $C_{i,j}$ indicam o elemento na i-ésima linha e j-ésima coluna. Deseja-se calcular a operação $C = A * B$, definida como

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \tag{2.13}$$

Para calcular a matriz C inteira deve-se utilizar a fórmula Equação 2.13 para todo par (i, j). Logo, pode-se definir um algoritmo básico para a multiplicação de matrizes como:

1: **for** $i = 1, n$ **do**
2: **for** $j = 1, n$ **do**
3: $C_{i,j} \leftarrow 0$
4: **for** $k = 1, n$ **do**
5: $C_{i,j} \leftarrow C_{i,j} + A_{i,k} * B_{k,j}$
6: **end for**
7: **end for**
8: **end for**

Analisando a complexidade desse algoritmo, tem-se que a linha 5 será executada n^3 vezes, já que ele está aninhada à 3 fors onde cada um executa n vezes. Logo, sabe-se que a complexidade desse algoritmo é $O(n^3)$.

2.3 Grafos

Um grafo pode ser definido como

$$G = (V, E)$$

onde V é um conjunto de vértices do grafo e $E = \{(x, y) : x \in V, y \in V\}$ um conjunto de arestas. Grafos podem ser representados graficamente como um conjunto de pontos, os vértices e linhas, as arestas, ligando os vértices.

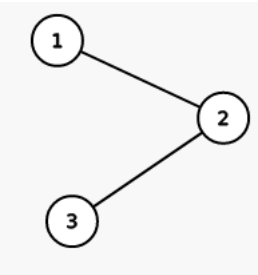

Figura 2.7: Grafo não direcionado.

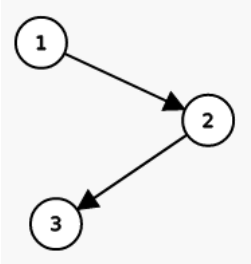

Figura 2.8: Grafo direcionado.

Na Figura 2.7 pode-se achar um grafo não direcionado (no qual um vértice $(a, b) = (b, a)$), onde $V = \{1, 2, 3\}$ e $E = \{(1, 2), (2, 3)\}$.

Já na Figura 2.8 pode-se achar um grafo direcionado (no qual um vértice $(a, b) \neq (b, a)$) onde $V = \{1, 2, 3\}$ e $E = \{(1, 2), (2, 3)\}$.

Além disso, pode-se definir um grafo com pesos no qual para cada vértice

$$e \in E$$

está associado um número real $w(e)$, chamado de peso dessa aresta. A Figura 2.9 mostra um grafo com pesos no qual a aresta $(1, 2)$ possui peso 3 e a aresta $(2, 3)$ peso 5.

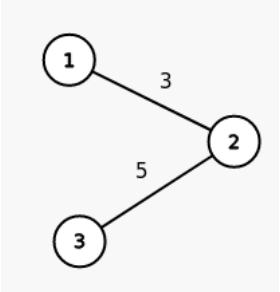

Figura 2.9: Grafo com pesos.

Se uma aresta (a, b) pertence ao grafo então diz-se que b faz parte do conjunto de vizinhos de a. Caso o grafo seja não direcionado então o oposto também é válido.

Grafos podem ser utilizados em diversas aplicações e, por isso, são bastante estudados na literatura.

2.3.1 Matriz de Adjacências

Há 2 abordagens comuns para representar grafos, as quais podem ser utilizadas para grafos direcionados e não direcionados, com peso e sem peso.

A primeira abordagem consiste na criação de uma matriz M, n x n, onde $n = |V|$, para a qual o valor $M_{i,j}$ informa a respeito da aresta (i, j), isto é, se $M_{i,j} = 0$, então **não** existe a aresta (i, j) no grafo, caso contrário ($M_{i,j} \neq 0$) então a aresta (i, j) **existe** no grafo e seu peso é $M_{i,j}$.

Em um grafo não direcionado, tem-se que $M_{i,j} = M_{j,i}$. Em um grafo sem pesos, ou $M_{i,j} = 0$ ou $M_{i,j} = 1$. A Figura 2.10 demonstra uma matriz de adjacências.

	1	2	3
1	0	3	0
2	3	0	5
3	0	5	0

Figura 2.10: Matriz de adjacências do grafo da Figura 2.9.

2.3.2 Lista de Adjacências

A segunda abordagem consiste na criação de uma vetor de listas L, de tamanho n, tal que para o vértice i do grafo, $L[i]$ contem uma lista com todos os nós j, tal que $(i, j) \in E$. Assim, se (a, b) não está em $L[a]$, então essa aresta não faz parte do grafo.

Em um grafo não direcionado, tem-se que $j \in L[i] \iff i \in L[j]$. Em um grafo com pesos, a lista $L[i]$ guarda tanto os vértice para os quais i, quanto a valor v dessa aresta. Em outras palavras: $L[2] = (1, 3), (3, 5)$ indica que existem as arestas: $(2, 1)$ com peso 3 e $(2, 3)$ com peso 5. A Figura 2.11 demonstra uma lista de adjacências.

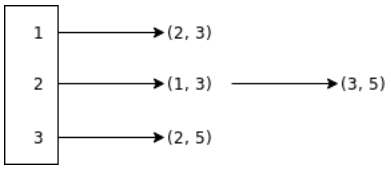

Figura 2.11: Lista de adjacências do grafo da Figura 2.9.

2.3.3 Dijkstra

Em um grafo, pode-se definir um caminho entre 2 vértices, v e u, como uma sequência de vértice

$$P = (a_0, a_1, ..., a_n)$$

tal que $a_0 = v, a_n = u, (a_{i-1}, a_i) \in E, 0 < i \leq n$ e $a_i \neq a_j, i \neq j$

Se esse grafo for com pesos, a peso desse caminho é definido como a soma dos pesos de suas arestas, ou seja, $w(P) = \sum_{k=1}^{n} w((a_{i-1}, a_i))$. Além disso, a distância entre 2 vértices v e u é definida como o menor peso entre todos os caminhos P entre v e u, a qual utilizará a notação $d(v, u)$.

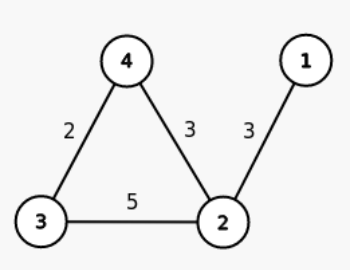

Figura 2.12: Distância entre 2 vértices.

No grafo da Figura 2.12, $P = (1, 2, 4)$ é um caminho entre 1 e 4 com peso $w(P) = 3+3 = 6$, o qual é a distância entre 1 e 4, já que o outro caminho existente, $P_2 = (1, 2, 3, 4)$ possui $w_{P_2} = 3 + 5 + 2 = 10$.

O problema de calcular a menor distância entre 2 vértices dados em um grafo com pesos é chamado de problema do menor caminho e pode ser resolvido com alguns algoritmos como Dijkstra e Bellman-Ford, os quais são capazes de resolver a versão mais genérica do problema chamada Single-Source Shortest Path (SSSP).

A ideia básica do algoritmo de Dijkstra [17] é construir uma árvore de menor caminho a partir de um nó inicial v. Sempre que um novo nó x é adicionado a árvore (um processo que será explicado em seguida), as arestas que saem desse nó são consideradas como uma possibilidade para a criação de um novo caminho, o qual consiste em utilizar a árvore para chegar no nó x e então utilizar a aresta que está sendo considerada, seja ela (x, a).

Como a árvore contém o menor caminho para o nó x, então esse caminho pode ser representado somente pela sua última aresta (x, a) e o peso desse caminho será $d(v, x) + w((x, a))$. Esse caminho é adicionado a uma estrutura de dados que contém, para cada vértice ainda não adicionado à árvore de menor caminho, o melhor caminho (com o melhor peso) achado com chegada nele. Dependendo da implementação, essa estrutura pode conter outros caminhos, os quais devem ser ignorados (veja linha 14).

Para a escolha do novo nó x a ser adicionado, o caminho entre os guardados na estrutura de dados que possuir menor peso será o menor caminho para o seu vértice de chegada i (cuja prova não será mostrada nesse trabalho). Assim, i é adicionado a árvore.

Considerando que a árvore inicia com o nó v, o qual possui $d(v, v) = 0$, segue o algoritmo abaixo:

1: Seja H a estrutura de dados mencionada acima, a qual guarda para cada elemento w o peso $w.d$ do caminho, e a última aresta $w.e = (x, a)$
2: Seja T a árvore de menor caminho
3: **for** $x \in V$ **do**
4: $\quad d(v, x) \leftarrow \infty$
5: **end for**
6: $d(v, v) \leftarrow 0$
7: v é a raiz de T
8: Adiciona $w.d = d(v, v) = 0$ e $w.e = (\phi, r)$ em H
9: **while** H não está vazio **do**
10: $\quad$ Encontra o elemento w com menor d em H
11: $\quad dist \leftarrow w.d$
12: $\quad (a, b) \leftarrow w.e$
13: $\quad$ Retira w de H
14: $\quad$ **if** $b \in T \vee dist > d(v, b)$ **then**
15: $\quad\quad$ **continue** ▷ Esse caminho não segue as condições definidas para H
16: $\quad$ **end if**
17: $\quad$ O pai de b em T é a
18: $\quad$ **for** $r \in V$ tal que $(b, r) \in E$ **do**
19: $\quad\quad$ **if** $d(v, b) + w(b, r) < d(v, r)$ **then**
20: $\quad\quad\quad d(v, r) \leftarrow d(v, b) + w(b, r)$
21: $\quad\quad\quad$ Adiciona $w.d = d(v, b)$ e $w.e = (b, r)$ em H
22: $\quad\quad$ **end if**
23: $\quad$ **end for**
24: **end while**

Analisando a complexidade do algoritmo acima, percebe-se, na inicialização, o comando da linha 3 é executado $O(|V|) = O(n)$ vezes. Além disso, o *while* da linha 9 será executado em tempo $O(p)$, tal que p é a quantidade de elementos em H ao longo de toda a execução do programa.

Assim, nas linhas 10 e 13, o tempo de execução será $O(p * (t_{min} + t_{del}))$.

A linha 18 possui um tempo de execução dependente da estrutura utilizada para representar o grafo. Caso tenha sida utilizada uma matriz de adjacências, então gasta um tempo $O(n)$ para achar os vizinhos de b. Caso tenha utilizada uma lista de adjacências, então gasta-se $O(|viz(b)|)$, onde $|viz(b)|$ é a quantidade de vizinhos do vértice b. Devido à retirada de elementos inadequados na linha 14, tem-se que a linha 18 é executada no

máximo uma vez para cada vértice do grafo ($O(n)$).

Logo, na linha 18, caso tenha sido utilizada uma matriz de adjacências, então gasta-se um tempo total de $O(n^2 * t_{ins})$. Caso tenha sido utilizada uma lista de adjacências, então gasta-se $O(\sum_{e \in E} |viz(e)| * t_{ins}) = O(|E| * t_{ins}) = O(m * t_{ins})$.

Onde t_{ins}, t_{min} e t_{del} são, respectivamente, o tempo de inserção de elemento, busca do mínimo e deleção do mínimo na estrutura de dados.

Assim, a complexidade do Dijkstra é $O(n + p * (t_{min} + t_{del}) + n^2 * t_{ins})$, se for utilizada matriz de adjacências, ou $O(n + p * (t_{min} + t_{del}) + m * t_{ins})$, se for utilizada lista de adjacências.

Algoritmo O(n^2)

A implementação mais simples da estrutura de dados é que ela seja um vetor, onde o índice i contêm as informações sobre o melhor caminho (de menor peso) que chega em i.

Assim, uma inserção nessa estrutura é só uma comparação e uma (possível) inserção na posição i e logo é $O(1)$.

A deleção é só invalidar a posição i e logo é $O(1)$. Já a busca pelo mínimo requer passar por todos os elementos do vetor para buscar o menor, sendo $O(n)$.

Além disso, p, a quantidade de elementos em H é n, já que o vetor possui um valor para cada vértice.

Assim, a complexidade do Dijkstra é $O(n + n * (n + 1) + n^2 * 1) = O(n^2)$, se for utilizada matriz de adjacências, ou $O(n + n * (n + 1) + m * 1) = O(n^2 + n * m) = O(n^2)$, se for utilizada lista de adjacências.

Binary Min Heap

Como discutido na Seção 2.2.4, tem-se:

$t_{ins} = O(lg(n))$, $t_{min} = O(1)$ e $t_{del} = O(lg(n))$.

Além disso, p, a quantidade de elementos em H pode ser até m, caso a condição da linha seja verdadeira em toda iteração do for.

Assim, a complexidade do Dijkstra é $O(n + m * (lg(n) + 1) + n^2 * lg(n)) = O(n^2 * lg(n))$, se for utilizada matriz de adjacências, ou $O(n + m * (lg(n) + 1) + m * lg(n)) = O(m * lg(n))$, se for utilizada lista de adjacências.

Fibonacci Heap

Como discutido na Seção 2.2.5, tem-se:

$t_{ins} = O(1)$, $t_{min} = O(1)$ e $t_{del} = O(lg(n))$.

Além disso, p, a quantidade de elementos em H pode ser até m, caso a condição da linha seja verdadeira em toda iteração do for.

Assim, a complexidade do Dijkstra é $O(n + m * (lg(n) + 1) + n^2 * 1) = O(n^2)$, se for utilizada matriz de adjacências, ou $O(n + m * (lg(n) + 1) + m * 1) = O(m * lg(n))$, se for utilizada lista de adjacências.

2.3.4 Floyd-Warshall

O algoritmo de Floyd-Warshall [18], [19] resolve um problema mais genérico que o Djikstra, que é encontrar o menor caminho de todo nó para todo nó.

Para esse algoritmo assume-se que se está utilizando uma matriz de adjacências M.

Inicialmente, só conhecem-se caminhos diretos entre vértices, onde direto é um caminho com 1 só aresta ou seja $p = \{a, b\}$. Em cada loop do algoritmo permitem-se o uso de um novo nó intermediário no caminho, onde um nó intermediário de um caminho $p = \{v_1, v_2, ..., v_r\}$ um nó v_i, tal que $i \neq 0$ e $i \neq r$.

Prova-se que caso sejam-se permitidos os nós intermediários na ordem $1, 2, ..., n$ e em uma etapa i utilizem-se os caminhos encontrados na etapa $i - 1$, ao final da etapa n encontra-se o menor caminho de todo nó para todo nó. Mais especificamente, a iteração i do nosso algoritmo encontrará, entre quaisquer dois nós, o caminho de menor peso que permite como nós intermediários **somente** os nós de 1 a i.

Para um par a, b qualquer de nós no grafo, seja $p_i = \{v_1, v_2, ..., v_r\}$ o caminho de menor peso entre a e b que permite como nós intermediários **somente** os nós de 1 a i. Perceba que p_i possui a seguinte relação com p_{i-1}:

- Se $i \notin p_i$, então p_i só contêm nós internos de 1 a $i - 1$ e logo $p_i = p_{i-1}$
- Se $i \in p_i$ e seja ele v_k, para $1 \leq k \leq n$. Então o caminho p_i pode ser divido em 2 caminhos q e s, tal que

  $q = \{v_1, v_2, ..., v_k\}$ e $s = \{v_k, v_{k+1}, ..., v_r\}$, ou seja, q é um caminho de a para i e s é um caminho de i para b.

  Como o caminho p é obtido seguindo o caminho q e logo em seguida o caminho s, então o peso w de p é $w(p) = w(q) + w(s)$

  Como o elemento i não pode estar 2 vezes no caminho p, uma vez que isso implicaria na existência de um ciclo, e não pode haver um ciclo em um caminho de menor peso, então $v_j \neq i, \forall j \neq k$.

  Logo, q e s só contêm nós internos de 1 a $i - 1$ e então já foram calculados na etapa $i - 1$.

Ao final da etapa n, será encontrado, entre quaisquer dois nós, o caminho de menor peso que permite como nós intermediários os nós de 1 a n, ou seja, os caminhos que permitem qualquer nó como caminho intermediário e logo o caminho com menor peso em todo grafo. Segue o algoritmo de Floyd-Warshall:

1: Seja $dist[1..n][1..n]$ a menor distância encontrada entre os vértices e seja $inter[1..n][1..n]$ o último nó intermediário utilizado nesse caminho (essa matriz é utilizada na impressão de todos os vértices do caminho).
2: **for** $i \leftarrow 1, n$ **do**
3: **for** $j \leftarrow 1, n$ **do**
4: **if** $M[i][j] > 0$ **then**
5: $dist[i][j] \leftarrow M[i][j]$
6: **else**
7: $dist[i][j] \leftarrow \infty$
8: **end if**
9: **end for**
10: **end for**
11: **for** $i \leftarrow 1, n$ **do**
12: **for** $j \leftarrow 1, n$ **do**
13: **for** $k \leftarrow 1, n$ **do**
14: **if** $dist[i][k] + dist[k][j] < dist[i][j]$ **then**
15: $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$
16: $inter[i][j] \leftarrow k$
17: **end if**
18: **end for**
19: **end for**
20: **end for**

A análise da complexidade do algoritmo de Floyd-Warshall pode ser feita analisando as linhas 4 e 14. A linha 4 será executada $O(n^2)$ vezes, uma vez que é executada para todo par (i, j), com $1 \leq i \leq n$ e $1 \leq j \leq n$.

Analogamente, a linha 14 será executada $O(n^3)$ vezes, já que é executada para toda tripla (i, j, k), com $1 \leq i \leq n$, $1 \leq j \leq n$ e $1 \leq k \leq n$.

Assim, a complexidade total do Fibonacci Heap é $O(n^2 + n^3) = O(n^3)$.

2.4 Resumo Conclusivo

Este capítulo apresentou os conceitos, estruturas de dados e algoritmos utilizados neste trabalho. Inicialmente, foram apresentadas as ferramentas utilizadas para a análise

desses algoritmos e, em seguida, foram apresentados os próprios algoritmos, com suas análises assintóticas.

Segue abaixo, tabelas que resumem os algoritmos e suas complexidades. Os algoritmos que possuem o mesmo conjunto de operações foram agrupados em uma mesma tabela para facilitar a comparação das implementações.

Tabela 2.1: Tabela resumo do algoritmo da Seção 2.2.1

	Atribuição	Referência	Inserção/Remoção no meio do array	Adição de um elemento no final
vetor	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Resizable-Array	$O(1)$	$O(1)$	$O(n)$	$O(1)$

Tabela 2.2: Tabela resumo do algoritmo da Seção 2.2

	Busca	Mínimo	Máximo	Inserção	Deleção
Vetor ordenado	$O(lg(n))$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Binary Search Tree	$O(lg(n))$	$O(lg(n))$	$O(lg(n))$	$O(lg(n))$	$O(lg(n))$

Tabela 2.3: Tabela resumo dos algoritmos da Seção 2.2.4 e da Seção 2.2.5

	Mínimo	Adição	Remoção
Vetor	$O(n)$	$O(1)$	$O(1)$
Min Heap	$O(1)$	$O(lg(n))$	$O(lg(n))$
Fibonacci Heap	$O(1)$	$O(1)$	$O(lg(n))$

Tabela 2.4: Tabela resumo do algoritmo da Seção 2.2.6

Implementação original	$O(n^3 * lg(n))$
Multiplicação de Matrizes	$O(n^3)$

Tabela 2.5: Tabela resumo dos algoritmos da Seção 2.3.1 e da Seção 2.3.2

	Algumas operações	Outras operações
Matriz de Adj.	$O(n^2)$	$O(1)$
Lista de Adj.	$O(n + m)$	$O(m)$

Tabela 2.6: Tabela resumo do algoritmo da Seção 2.3.3

	Dijkstra
Vetor e Matriz de Adj.	$O(n^2)$
Min Heap e Matriz de Adj.	$O(m * lg(n) + n^2 * lg(n))$
Min Heap e Lista de Adj.	$O(m * lg(n))$
Fibonacci Heap e Matriz de Adj.	$O(m * lg(n) + n^2)$
Fibonacci Heap e Lista de Adj.	$O(m * lg(n))$

Tabela 2.7: Tabela resumo do algoritmo da Seção 2.3.4

Implementação original	$O(n^4)$
Floyd-Warshall	$O(n^3)$

Capítulo 3

Análise do Simulador ONS

Este capítulo tem como objetivo apresentar o simulador sobre o qual o trabalho foi realizado, explicando seu funcionamento e seu modo de utilização.

3.1 Motivação

O rápido crescimento do uso da Internet e do consumo de banda leva a uma necessidade da existência de uma infraestrutura de rede robusta e capaz de suprir altas demandas. Nesse contexto, surgem as redes ópticas elásticas como uma opção para prover esse serviço.

Soluções de problemas relacionados à EON devem ser avaliadas através de simulações, visto que sua alta complexidade dificulta a criação de um modelo matemático e que a implementação em um ambiente real seria cara e não há disponibilidade da tecnologia.

Devido à isso, Costa, L. R. et al. desenvolveram um simulador para avaliação de redes ópticas [1] escrito em Java, utilizando eventos discretos para a execução de chamadas de tráfego dinâmicas e que permite o uso de WDM ou de EON.

3.2 Redes Ópticas Elásticas

As redes ópticas elásticas surgiram como uma solução para melhorar o funcionamento das redes ópticas, as quais tradicionalmente utilizam WDM.

A tecnologia de multiplexação por divisão de comprimento de onda (WDM) divide o espectro em vários canais separados distanciados por 50 ou 100 GHz [20]. Devido a essa alta largura do canal, caso esteja-se passando um sinal com baixa largura de

banda, um grande intervalo de frequências pode ficar inutilizado e essa tecnologia não possui uma ferramenta para a utilização dessa faixa.

Com o objetivo de tratar essa questão e assim reduzir o intervalo de frequências não utilizados, foram propostas as redes ópticas elásticas. Nela, existem canais de baixa largura de banda chamados de slots, que podem ser alocados em conjuntos de tamanho arbitrário através da técnica OFDM (do inglês Orthogonal Frequency Division Multiplexing) e cuja modulação pode ser escolhida independente de outras conexões. Logo, para atender uma demanda, consegue-se utilizar a quantidade mínima de slots necessários, reduzindo a inutilização de recursos.

Deve-se notar porém que devido a tecnologia física por trás, esses slots devem ser alocados de uma maneira contígua (na frequência) e, via de regra, uma demanda irá criar um caminho óptico pela rede na qual essas frequências devem ser igual, ou seja, se em um link foram alocadas 4 *slots* a partir de uma frequência x, em todo os links desse caminho devem ter sido alocados 4 *slots* a partir da mesma frequência x.

Essas condições dificultam a alocação de recursos em uma rede óptica elástica, uma vez que elas criam situações nas quais há banda o suficiente na rede para atender a demanda, mas não é possível atendê-la.

- Quando há fragmentação em um link:

  Imagine que há 10 slots em um link e 2 demandas estão sendo atendidas, uma está alocada nos slots de 3 a 5 e a outra está alocada nos slots 9 e 10. Assim os intervalos $[1, 2]$ e $[6, 8]$ estão livres, ou seja, há 5 slots livres.

  Porém caso chegue uma demanda por 4 slots ela não pode ser atendida já que não existem 4 slots contíguos.

- Quando não é possível criar um caminho contínuo:

  Assumindo 3 nós, 1 2 e 3, o nó 1 possui conexão com o 2 e o nó 2 possui conexão com 3. Caso hajam 10 slots na fibra e na conexão $(1, 2)$ está livre o intervalo $[1, 6]$ e na conexão $(2, 3)$ está livre o intervalo $[5, 10]$. Caso chegue uma demanda de 4 slots levando de 1 a 3, ela não poderá ser atendida, visto que não existem 4 slots livres em ambas as conexões, somente o intervalo $[5, 6]$ que possui somente 2 slots.

Essas características de uma rede EON fazem com que, em comparação com redes WDM, sejam necessários mecanismos mais complexos para a alocação de recursos na rede, o que cria o problema de *Routing and Spectrum Assignment* (RSA) [21], onde deve-se alocar uma conexão com banda suficiente para uma demanda de x bytes de um nó de origem à um nó de destino.

Esse problema pode ser modificado para incluir a escolha do nível de modulação do sinal, o que se denomina *Routing, Modulation Level, and Spectrum Allocation* (RMLSA) [22].

Esses problemas, RSA e RMLSA, são problemas NP-completos [4] e, portanto, bastante estudados na literatura, levando a várias abordagens que tentam melhorar a eficiência da alocação e para verificá-las viu-se necessária a criação de simuladores que possam analisar o desempenho dessas abordagens [1], [23], [24].

Em [25], foram sugeridos os algoritmos RSA:

KSP, o qual utiliza uma abordagem de 2 passos, onde no 1º passo, é utilizado o algoritmo de Yen para o problema do *k-shortest paths* e no 2] passo, os caminhos são processados procurando-se alocar a demanda a um deles.

MSP, o qual utiliza um algoritmo de Djikstra modificado para encontrar o menor caminho.

Posteriormente, em [26], foi estudado o uso de modulação adaptativa em algoritmos RSA e foram propostos os algoritmos *m Adaptive RSA algorithms*, chamados mAdap, os quais iteram entre os modulações possíveis que podem ser aplicadas ao sinal, em ordem decrescente, até encontrar uma solução [21].

Assim, aplicando a modulação adaptativa nos algoritmos de [25], são propostos os algoritmos *mAdapKSP* e *mAdapMSP*.

3.3 Simulador

3.3.1 Funcionamento

O funcionamento do simulador está dividido em 4 etapas:

1. A configuração do ambiente de execução

   Através da leitura de um arquivo XML, carregam-se informações configuráveis referentes a execução atual, as quais são:

   - O tipo de simulador WDM ou EON
   - O algoritmo de roteamento e alocação desejado, como por exemplo, RSA, RMLSA, sendo possível a criação de um esquema próprio de roteamento através de uma nova classe Java
   - Dados a respeito do tráfego e das chamadas executadas na rede.
   - O topologia física do grafo, a qual representa a rede óptica.

2. Geração dos eventos a serem executados pelo simulador

   O simulador gera as chamadas a partir do parâmetro *seed* (explicado na Seção 3.3.2) dada na chamada de execução. Essas chamadas possuem um nó de partida, um nó de destino, uma quantidade de banda requerida e o tempo de duração.

   Ao final, esses eventos serão colocados em uma fila de prioridades, ordenada pelo momento de início.

3. Execução da simulação

   Para cada evento retirado da fila de prioridades, executa-se o algoritmo de roteamento e alocação obtido na 1ª etapa, o qual toma a desição de aceitar ou rejeitar uma chamada. Nessa etapa, todas as ações são gravadas em um arquivo de trace.

4. Avaliação dos resultados

   Fornece as ações gravados no trace e provê uma interface que permite a adição de estatísticas específicas requeridas na aplicação executada.

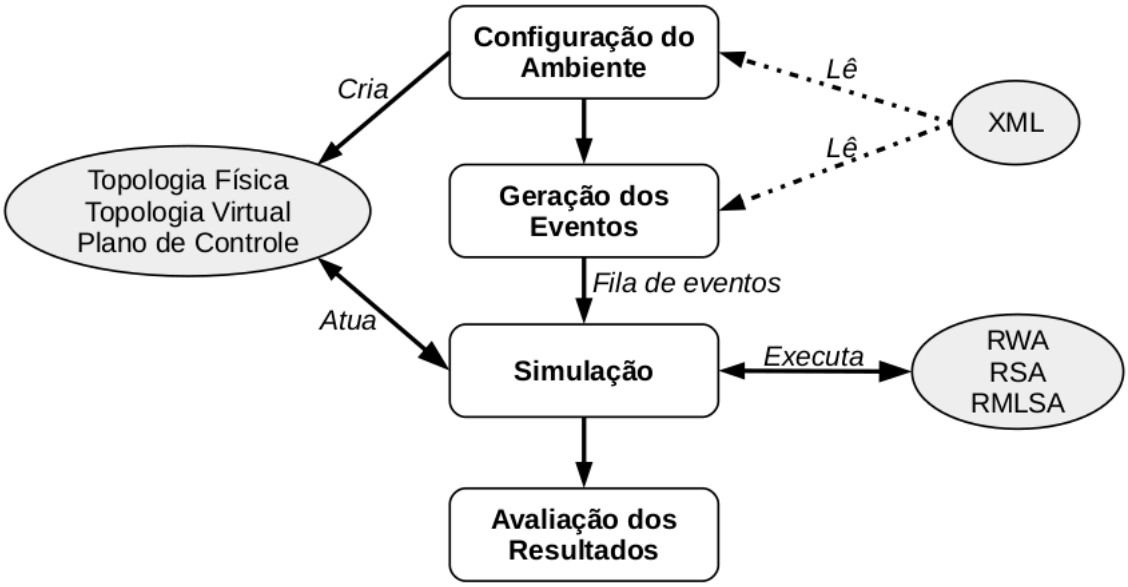

Figura 3.1: Modelo do funcionamento do simulador, imagem adaptada de [1].

3.3.2 Execução

Para executar o simulador, os seguintes parâmetros são utilizados:

- *simulation_file*, parâmetro **obrigatório**, contêm a localização do arquivo XML com as configurações do ambiente.
- *seed*, parâmetro **obrigatório**, aceita os valores entre 1 e 25, os quais equivalem a 25 boas sementes definidas pelo simulador.
- *trace*, parâmetro *opcional*, ativa o arquivo de trace
- *verbose*, parâmetro *opcional*, para geração de mensagens a respeito do funcionamento do simulador na saída padrão.
- *−l load*, parâmetro *opcional*, seleciona a carga desejada de execução.
- *−L minload maxload step*, parâmetro *opcional*, executa o programa com diversas cargas, começando em *minload*, acrescendo de *step* até executar pro último *maxload*.

3.3.3 Estrutura de Diretórios

Os arquivos do simulador estão divididos nos seguintes diretórios segundo sua funcionalidade:

Tabela 3.1: Tabela da estrutura do simulador

Diretório	Arquivo	Descrição
xml		Contêm os arquivos xml de configuração.
	simulation-COMNET	Contêm as configurações utilizadas para a simulação do artigo [22].
ra		Contêm os códigos dos algoritmos de roteamento e alocação.
	ControlPlaneForRA	Essa interface provê vários métodos para a classe RWA no ControlPlane.
	RA	Essa interface provê vários métodos para as classes RA.
	MAdapKSP	Um dos algoritmos para o problema RSA utilizado no artigo [22], utiliza a algoritmo de *K shortest paths*.

Diretório	Arquivo	Descrição
	MAdapMSP	Um dos algoritmos para o problema RSA utilizado no artigo [22], utiliza a algoritmo de *Dijsktra*.
tools		Contêm ferramentas para carregar e salvar objetos.
	LoadObject	Ferramenta para carregar um objeto.
	SaveObject	Ferramenta para salvar um objeto.
utils		Contêm algoritmos úteis para o funcionamento geral do simulador.
	AllPaths	Contêm um objeto estático com todos os caminhos de qualquer nó até qualquer nó que possuem um peso menor ou igual a um valor dado, utiliza a classe KSPOffline para obter os caminhos.
	AllPathsNode	Contêm os caminhos dados em um arquivo de texto de qualquer nó até qualquer nó que não possuem repetição de nós.
	ComputeSNR	Implementa vários métodos, definidos em diferentes trabalhos, para computar o SNR (do inglês signal-to-noise ratio).
	Dijkstra	Implementa o arquivo de Dijkstra para encontrar o menor caminho de um nó de origem a um nó de destino.
	Distribution	Implementa uma boa função para obter sequências de números aleatórios. Utilizado junto ao parâmetro *SEED* da linha de comando.

Diretório	Arquivo	Descrição
	DSP	Obtêm todos os menores caminhos de um nó de origem a um nó de destino, tal que esses caminhos não possuam nenhuma aresta em comum (edge disjoint shortest pair).
	Edge	Implementa a classe Edge, que define uma aresta do grafo.
	GraphStatus	Permite a reserva de fluxos no grafo para uma demanda específica.
	KSPOffline	Executa uma vez e salva o resultado do algoritmo de K shortest paths, de forma que posteriormente seja necessário somente o retorno de uma posição numa matriz para obter o resultado.
	LayeredGraph	Constrói um grafo "layered", o qual conecta 2 layers (camadas) diferentes, cada qual é um grafo por si só.
	MetricFragmentation	Implementa vários métodos, definidos em diferentes trabalhos, para computar a fragmentação de um link.
	MultiWeightEdge	Implementa a classe Edge, que define uma aresta do grafo que pode ter mais de um peso.
	PseudoControlPlane	Utilizada por classes RA para controlar a grafo.
	PseudoControlPlane2	Utilizada por classes RA para controlar a grafo, adicionando suporte a algumas funcionalidades extras.
	WeightedGraph	Implementa um grafo com pesos, utilizando uma matriz de adjacências.

Diretório	Arquivo	Descrição
	WeightedGraphMultiWeight	Implementa um grafo onde uma aresta pode ter vários níveis, cada um com seu peso, utilizando uma matriz de adjacências.
	WeightedGraphMultiWeightEdge	Implementa uma aresta da classe WeightedGraphMultiWeight, onde há diversos níveis cada um com o seu peso.
	WeightedGraphMultiGraph	Implementa um grafo onde entre dois vértice *a* e *b* podem ter várias arestas, cada um com seu peso, utilizando uma lista de adjacências.
	WeightedGraphMultiGraphMultiWeight	Implementa um grafo onde entre dois vértice *a* e *b* podem ter várias arestas, cada podendo ter vários pesos, utilizando uma lista de adjacências.
	YenKSP	Implementa o algoritmo de Yen para obter os k menores caminhos sem loop entre um nó de origem e um nó de destino.
ons		Código do simulador em si, utiliza o diretório utils para algumas funcionalidades.
	ArgParsing	Classe responsável por fazer o *parse* dos argumentos passados pela linha de comando.
	ControlPlane	Classe responsável por controlar os recursos e conexão dentro da rede.
	EONLightPath	Provê um lightpath para unir tráfego de vários links com números diferentes de slot.
	EONLink	Representa um link em uma rede óptica dividido em vários slots.

Diretório	Arquivo	Descrição
	EONOXC	Se refere a um OXC utilizado em uma rede óptica elástica que lida com modularização.
	EONPhysicalTopology	Objeto que representa a topologia física.
	Event	Objetos de Event só possuem um atributo: scheduled time.
	EventScheduler	A linha do tempo do simulador não é marcada pelo tempo de relógio, mas por eventos.
	Flow	A classe Flow define um objeto que pode ser pensado como um fluxo de dados, indo de um nó de origem a um nó de destino.
	FlowArrivalEvent	Métodos para tratar a chegada de um objeto Flow.
	FlowDepartureEvent	Métodos para tratar a saída de um objeto Flow.
	LightPath	Em uma rede óptica, um lightpath é um caminho óptico que pode incluir vários links em uma rede.
	Link	A classe abstrata Link.
	Main	A classe Main cuida da execução do simulador, o que inclui lidar com os argumentos chamados (ou não) na linha de comando.
	Modulation	O container da classe Modulation.
	MyStatistics	Essa classe calcula e imprime todas as estatísticas do simulador.
	OrdinaryEvent	A classe OrdinaryEvent.
	OXC	Os Optical Cross-Connects (OXCs) estão presentes nos nós, para roteamento do tráfego de dados e possui grooming input e output ports.
	PhysicalImpairments	A classe PhysicalImpairments calcula o SNR de lightpaths.

Diretório	Arquivo	Descrição
	Path	Um caminho é simplesmente uma lista de lightpaths.
	PhysicalTopology	A topologia física de uma rede se refere ao layout físico dos dispositivos na rede ou a maneira na qual os dispositivos da rede estão organizados e como eles se comunicam um com os outros.
	SimulationRunner	Roda a simulação enquanto houverem eventos a serem executados.
	Simulator	Centraliza a execução do simulador.
	Tracer	Cria um arquivo de trace que contem informações sobre o que ocorreu durante a simulação, tais como fluxos de chegada e de saída, lightpaths criados, e fluxos aceitados e bloqueados.
	TrafficGenerator	Gera o tráfego da rede baseado na informação passada através dos argumentos da linha de comando e do arquivo XML da simulação.
	TrafficInfo	Retorna informações detalhadas a respeito do tráfego da rede: holding time, rate, classe de serviço e peso.
	VirtualTopology	A topologia virtual é criada baseada em um Physical Topology dado e nos lightpaths especificados no arquivo XML.
	WDMLightPath	O Wavelength Division Multiplexing (WDM) LightPath representa um lightpath em uma rede óptica WDM.

Diretório	Arquivo	Descrição
	WDMLink	O Wavelength Division Multiplexing (WDM) Link representa um link em uma rede óptica.
	WDMOXC	O WDM Optical Cross-Connects (EDMOXCs) consegue trocar o sinal óptico chegando em um comprimento de onda do link óptico de entrada para o mesmo comprimento de onda no link de saída.
	WDMPhysicalTopology	O objeto de topologia física WDM.

3.4 Metodologias de Avaliação de Desempenho

Há três métodos de avaliação de um sistema: *Modelagem analítica*; *simulação* e *medição* [27]. A escolha de qual método será utilizado é feita através da análise de vários parâmetros, entre os quais constam:

- Estágio, deve-se considerar em qual estágio do ciclo de vida do produto se está. Para o método da *medição*, é necessário a existência prévia de um sistema (real) para o problema, enquanto as outras metodologias podem ser utilizadas em qualquer estágio.
- Ferramentas, deve-se considerar quais são as ferramentas necessárias para implementar esse método. *Modelagem analítica* requer um analista com habilidades de modelagem e de análise de sistemas. *Simulação* requer conhecimento de linguagens de computação e técnicas de simulação. *Medição* requer a instrumentos de medida do produto, ou seja, um sistema real para medir o que se deseja.
- Custo, deve-se considerar o custo da implementação do método. *Medição* requer equipamentos e instrumentos reais e por isso é a mais custosa das alternativas. Já a *modelagem analítica* requer somente o tempo do analista, papel e lápis, sendo então a alternativa mais barata. A *simulação* é então uma alternativa de custo médio.

Para poder avaliar soluções dos problemas relacionados à EON é necessário o uso de *simulações*, uma vez que não é possível a criação de um *modelo matemático* devido a sua alta complexidade que impossibilita a modelagem do sistema e também não se

consegue realizar uma *medição* em um ambiente real devido ao custo e disponibilidade da tecnologia [7].

Além disso, para análise da performance desse sistema, deve-se escolher métricas para sua análise [27]. Nesse trabalho, trabalha-se com simulações que terminaram com sucesso, isto é, apresentam um resultado correto em um tempo finito e, como deseja-se analisar o tempo de execução do simulador, a métrica de performance utilizada é o tempo necessário para realizar o serviço e procura-se minimizá-lo.

Por fim, para complementar a metodologia, além de um método de avaliação e de uma métrica deve-se escolher um método para a análise do resultado. Nesse trabalho é utilizado o método de replicações independentes, o qual requer que a simulação seja repetida várias vezes com *seeds* diferentes.

O uso de *seeds* diferentes garante que não haja correlação entre repetições diferentes e que, logo, a média e a variância obtidas pela Equação 3.1 e pela Equação 3.2 são válidas, o que é necessário para a análise dos resultados.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3.1}$$

$$\bar{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{3.2}$$

Obtidas a média e a variância dos testes feitos, consegue-se plotar os resultados e analisá-los.

3.5 Análise de Oportunidades

Para poder analisar possível melhorias no simulador, foi estudado o código, em especial o diretório *utils*, o qual possui a coleção de algoritmos clássicos utilizados no programa. Nesse estudo, foram-se observados as seguintes oportunidades:

3.5.1 Lista de Adjacências

Na classe *WeightedGraph*, assumindo um grafo com n vértices e m arestas, pode ser modificada a estrutura de dados que representa o grafo de uma matriz de adjacências para uma lista de adjacências, o que implica em uma redução na complexidade assintótica de $O(n^2)$ para $O(n+m)$ para certas funções e um aumento de $O(1)$ para $O(m)$ para outras funções. Será testado se essa redução em algumas funções compensa o aumento em outras.

3.5.2 Floyd-Warshall

Serão mudada as funções *getAveragePathLength* e *getGraphDiameter* para utilizar o algoritmo de Floyd-Warshall, o que reduz a complexidade de O(n^4), obtida através da execução do algoritmo de Dijkstra n^2 vezes, para O(n^3). Após feitas as modificações, serão feitos testes para verificar o desempenho dessa proposta e ela será comparada com a versão atual.

3.5.3 Multiplicação de Matrizes

Após a modificação da estrutura de dados que representa o grafo (Lista de Adjacências), pode-se modificar *getClusteringCoefficient* para manter a complexidade em $O(n^2)$.

Além disso, em *getGlobalClusteringCoefficient* é possível reduzir a complexidade de $O(n^3 * log(n))$ para $O(n^3)$, através da multiplicação de matrizes.

3.5.4 Min Heap e Fibonacci Heap

Na classe *Dijkstra*, pode-se modificar a função que obtém o vértice de menor distância utilizando um min heap, o que reduziria a complexidade dessa função de $O(n)$ para $O(log(n))$ ou utilizando uma heap de Fibonacci, no qual operações de inserção ou buscar o menor elemento possuem complexidade $O(1)$ e a deleção possui complexidade $O(log(n))$.

Assim, a complexidade do Dijkstra como um todo é de:

- O(n^2), na implementação atual.
- O($m * log(n) + n^2 * log(n)$) com Min Heap e Matriz de Adjacências.
- O($m * log(n)$) com Min Heap e Lista de Adjacências.
- O($m * log(n) + n^2$) com Fibonacci Heap e Matriz de Adjacências.
- O($m * log(n)$) com Fibonacci Heap e Lista de Adjacências.

3.6 Descobertas através do Profiling

Utilizando o IDE Netbeans, executou-se o profiling do código, o qual utilizava a mesma configurações (xml/linha de comando) dos testes executados. Assim, consegue avaliar quais as seções de desempenho mais crítico em uma execução com configurações padrão de uso do simulador.

3.6.1 Resizable-Array

Ao executar o profiling, verificou-se que mais de 85% do tempo de execução era gasto na função *getSlotsAvailableToArray* da classe *EONLink*, a qual chama a função *getSlotsAvailable*.

A função getSlotsAvailable (62.4% do tempo de execução) recebe um inteiro *requestedSlots* indicando a quantidade de slots requisitada e deve retornar **todas** as posições tais que é possível alocar para essa chamada *requestedSlots* slots a partir dessa posição sem que uma delas tenha sido utilizada por outra função e mantendo uma banda de guarda.

Para isso, essa função chama *areSlotsAvailable*, a qual verifica se pode ser feita uma alocação para essa chamada a partir de uma posição específica. Em seguida, caso o retorno desta função tenha sido verdadeiro, essa posição é adicionada a uma *TreeSet* do Java, o qual implementa uma binary search tree (Seção 2.2). Ao final, essa *TreeSet* é retornada a função *getSlotsAvailableToArray*.

Na função *getSlotsAvailableToArray*, os elementos da *TreeSet* são retirados 1 a 1 para serem adicionados a um array.

Sabendo que as operações de uma Binary Search Tree são executadas em tempo $O(lg(n))$ para n elementos e assumindo que *areSlotsAvailable* é executada em tempo $O(f(s))$, para $f(s)$ uma certa função de s, onde s é o número de *slots* existentes em um nó da topologia, então temos que:

A função *getSlotsAvailable* é executado em tempo $O(s * f(s) * lg(s))$ e a função *getSlotsAvailableToArray* em tempo $O(s * f(s) * lg(s) + s * lg(n))$.

É importante notar que o uso da Binary Search Tree possui como objetivo manter as posições ordenadas para que o array final esteja ordenado. Porém, como avaliam-se todas as posições em um loop que inicia de 1 até s então as posições já serão obtidas de forma ordenada, uma vez que a posição $i + 1$ só poderá ser adicionada à *TreeSet* após a posição i devido a ordem de execução do loop.

Isso permite a melhoria da complexidade através do uso da Resizable-Array, a qual possui complexidade assintótica $O(1)$ para a adição de elementos ao final do array e $O(1)$ para indexação, melhorando a complexidade para $O(s * f(s))$ na função *getSlotsAvailable* e $O(s * f(s) + s)$ na função *getSlotsAvailableToArray*.

Perceba que como não se conhece o tamanho total de posições que satisfazem a condição, caso fosse usado um array comum, seria necessário realocar a memória várias vezes, o que possui complexidade $O(s)$, o que levaria a uma complexidade final de $O(s^2 * f(s) + 1)$, o que não é muito eficiente.

3.6.2 Greedy Approach

Na discussão acima, não foi analisada a função *areSlotsAvailable*, responsável pela verificação se a posição pode ser alocada. Essa função verifica se as posições entre *begin* e *end* podem ser utilizadas para uma chamada garantindo uma banda de guarda antes e depois.

Essa função inicialmente verifica para todas as posições entre *begin* e *end*, se elas estão livres, então ela verifica se *guardband* posições antes e *guardband* posições depois estão livres ou já são banda de guarda. A complexidade dessa operação é $O((end - begin) + 2 * guardband) = O(requiredSlots + 2 * guardband)$, o que é no pior caso $O(s)$.

Perceba que se uma posição i pode ser alocada, então as posições i até $i+requiredSlots-1$ estão vazias e nos intervalos $[i - guardband, i - 1]$ e $[i + requiredSlots, i + requestedSlots + guardband - 1]$ estão livres ou ocupadas.

Assim para a posição $i + 1$, sabe-se que:

- O intervalo $[i + 1 - guardband, i]$ está livre ou reservado, ou seja, há uma banda de guarda para a esquerda.
- O intervalo $[i + 1, i + requiredSlots - 1]$ está livre e $i + requiredSlots$ está livre ou ocupado.

  Ou seja, **caso** $i + requiredSlots$ esteja livre, então o intervalo começando em $i + 1$ com *requiredSlots* slots está todo livre.
- O intervalo $[i + requiredSlots + 1, i + requestedSlots + guardband - 1]$ está livres ou ocupado.

  Ou seja, **caso** $i + requestedSlots + guardband$ seja livre ou reservada então há banda de guarda para a direita.

Assim, consegue-se aproveitar a computação de uma posição nas outras, permitindo a verificação de uma posição x em $O(1)$ ao invés de $O(requiredSlots + 2 * guardband)$. E assim, a verificação de **todas** as posições gastará um tempo de até $O(s)$.

Perceba que a verificação de todas as posições é feita pela função *getSlotsAvailable* e não pela *areSlotsAvailable*. Assim, a complexidade final de getSlotsAvailableToArray será $O(s + s) = O(s)$, utilizando essa abordagem e Resizable-Array, já que $f(n) = 1$. Enquanto somente com Resizable-Array, teria-se $O(s^2)$, já que $f(n) = O(requiredSlots + 2 * guardband) = O(s)$.

3.7 Resumo Conclusivo

Neste capítulo, foram explicados conceitos de redes ópticas elásticas, as quais conseguem ajustar dinamicamente a largura de banda dada à uma demanda devido, principalmente, à tecnologia OFDM. Porém, essa largura de banda deve ser alocada de forma contígua na frequência e formar um caminho contínuo, onde as frequências utilizadas para transmissão do dado são iguais, até o destino.

Isso permite um aproveitamento melhor da rede, porém dificulta o problema de roteamento e alocação de recursos (RSA).

Em seguida, foi explicado o funcionamento do simulador, o qual é dividido em 4 etapas: configuração do ambiente; geração dos eventos; simulação e avaliação dos resultados. O simulador permite a configuração de vários aspectos relacionados à análise de redes, como a topologia física e o algoritmo RSA a ser utilizado. Além disso, o simulador permite a passagem dos seguintes parâmetros pela linha de comando:

- *simulation_file*, parâmetro **obrigatório**, contêm a localização do arquivo XML com as configurações do ambiente.
- *seed*, parâmetro **obrigatório**, aceita os valores entre 1 e 25, os quais equivalem a 25 boas sementes definidas pelo simulador.
- *trace*, parâmetro *opcional*, ativa o arquivo de trace
- *verbose*, parâmetro *opcional*, para geração de mensagens a respeito do funcionamento do simulador na saída padrão.
- *−l load*, parâmetro *opcional*, seleciona a carga desejada de execução.
- *−L minload maxload step*, parâmetro *opcional*, executa o programa com diversas cargas, começando em *minload*, acrescendo de *step* até executar pro último *maxload*.

A seguir, foi explicado a respeito das metodologias de avaliação de desempenho existentes na área da computação e verificou-se que, no contexto de redes EON, é necessário a utilização de simulação como único método de avaliação de um sistema possível e viável.

Adicionalmente, escolheu-se como métrica de perfomance o tempo necessário para realizar o serviço (simulação), o qual será avaliado através do método de avaliação de performance: método de replicações independentes.

Por fim, foram explicadas as oportunidades encontradas no simulador, as quais utilizam a base teórica do Capítulo 2. Essas modificações estão apresentadas na Tabela 3.2 a seguir

Tabela 3.2: Tabela resumo do Capítulo 3

Oportunidade	Complexidade Original	Nova complexidade
Lista de Adjacências	$O(n^2)$ para algumas funções e $O(1)$ para outras	Redução de $O(n^2)$ para $O(n + m)$ para certas funções e um aumento de $O(1)$ para $O(m)$ para outras
Floyd-Warshall	$O(n^4)$	$O(n^3)$
Multiplicação de Matrizes	$O(n^3 * log(n))$	$O(n^3)$
Min Heap e Matriz de Adjacências	$O(n^2)$	$O(m * log(n) + n^2 * log(n))$
Min Heap e Lista de Adjacências	$O(n^2)$	$O(m * log(n))$
Fibonacci Heap e Matriz de Adjacências	$O(n^2)$	$O(m * log(n) + n^2)$
Fibonacci Heap e Lista de Adjacências	$O(n^2)$	$O(m * log(n))$
Resizable-Array	$O(s^2 * log(s))$	$O(s^2)$
Greedy Approach	$O(s^2 * log(s))$	$O(s)$

Capítulo 4

Análise dos Resultados

Para finalizar esse trabalho, foram testados os algoritmos propostos no simulador. Para esses testes, baseou-se no artigo de Costa, L. R. *et al* [22], o qual utilizou o simulador ONS para seus testes.

Para o teste foi utilizada uma máquina virtual Ubuntu versão 16.04 x86_64 com 2 CPUs e 2 MiB de memória RAM rodando em uma máquina real Fedora 27 com processadores Intel Core i7 de 2.7 GHz de frequência.

Mais especificamente, utilizou-se o mesmo arquivo xml de configuração, *simulation-COMNET.xml*, e os mesmos módulos RA, *MAdapKSP* e *MAdapMSP*, garantindo-se assim a mesma topologia e algoritmo de roteamento. Vale-se observar que isso segue o princípio de testar um caso real de uso do simulador.

Foram testados os algoritmos RA: *MAdapKSP*, o qual possui um tempo de execução curto para cada teste, e *MAdapMSP*, o qual possui um tempo de execução maior por teste. Esses algoritmos foram selecionados para o nosso teste porque o algoritmo principal que eles utilizam para auxiliar na escolha da alocação de recursos são, respectivamente, o algoritmo de Yen (problema do K-shortest paths) e o algoritmo de Djikstra, os quais utilizam para seu funcionamento as funções que foram modificadas neste trabalho, sendo assim interessantes para a nossa análise.

Foram utilizados 2 cenários de teste, que provêm dos 2 possíveis parâmetros utilizados na linha de comando para escolher a carga: *-l*, que executa somente 1 carga e *-L*, que executa uma faixa de cargas.

Esses cenários foram considerados porque quando estava-se planejando a fase de testes, percebeu-se que se fossem executadas várias cargas uma a uma e ao final somado o tempo e se fossem executadas todas as cargas em um único comando, ocorreu que um algoritmo era mais eficiente em um dos cenários e o outro era mais eficiente no outro e, portanto, decidiu-se considerar essa diferença na análise dos algoritmos.

As linhas de comandos utilizada para os testes foram:

/usr/bin/time -f "\t%E|%U|%S"java -jar MODIFICACAO.jar -f XML-PATH/simulation-COMNET.xml -s SEED **-l LOAD** -table > MODIFICACAO-RA-sSEED

O que equivale à um cenário onde somente 1 carga é utilizada em 1 execução, e:

/usr/bin/time -f "\t%E|%U|%S"java -jar MODIFICACAO.jar -f XML-PATH/simulation-COMNET.xml -s SEED **-L MINLOAD MAXLOAD STEP** -table > MODIFICACAO-RA-sSEED

O que equivale à um cenário onde uma faixa de cargas carga é utilizada em 1 execução, onde:

/usr/bin/time -f "\t%E|%U|%S" é uma modificação do comando time do Linux utilizada para modificar o formato de saída dos dados. Esse comando retorna o tempo real desde o início da execução até o fim, o tempo em segundos no qual a CPU executou em modo usuário e o tempo em segundos no qual a CPU executou em modo kernel, nessa ordem. Para obter-se o valor a ser utilizado na análise de resultados, deve-se somar os tempos de modo usuário e modo kernel, uma vez que eles dão o tempo total no qual o simulador foi executado na CPU.

Vale mencionar que o tempo real não deve ser utilizado, uma vez que devido ao sistema operacional permitir execução de vários processos na mesma máquina, o simulador pode permanecer um certo tempo sem ser executado, esperando que a CPU esteja disponível. Porém, o tempo real incluirá esse tempo no qual o simulador não esteja executando, já que ele mede a diferença do relógio do sistema entre o começo da execução e o seu término.

MODIFICACAO.jar é o jar da modificação sendo testada, por exemplo, no teste do uso da Fibonacci Heap, esse arquivo era *DF.jar*.

XML-PATH é o caminho para o arquivo XML na maquina local.

SEED é a seed sendo testada no momento.

RA é o algoritmo RA sendo utilizado pelo arquivo xml de configuração, ou seja, RA é *MAdapKSP* ou *MAdapMSP*.

LOAD é a load escolhido para ser executado, no caso de ter sido utilizada 1 carga por execução. Esse valor era (em execuções diferentes): 400, 440, 480, 520, 560 ou 600.

MINLOAD é a load mínimo a ser executado no cenário de várias cargas por execução. Seu valor é de 400.

MAXLOAD é a load máximo a ser executado no cenário de várias cargas por execução. Seu valor é de 600.

STEP é o quanto deve ser acrescentado a uma carga que se está executando para definir a próxima carga a ser executada. Seu valor é de 40.

Assim, com *MINLOAD* = 400, *MAXLOAD* = 600 e *STEP* = 40, tem-se que, no **cenário de várias cargas por execução**, são utilizadas as cargas: 400, 440, 480, 520, 560 e 600 em uma **única** linha de comando. Devido a isso, obtem-se como resultado um único valor de tempo que equivale a execução das 6 cargas acimas juntas.

Isso difere-se do **cenário de 1 execução por carga**, o qual utiliza as mesmas cargas: 400, 440, 480, 520, 560 e 600 **1 carga por cada linha de comando**. Assim, para testar todas as cargas, deve-se executar a linha de comando 6 vezes, alterando o parâmetro LOAD. E, devido a isso, obtem-se como resultado um gráfico com vários valores tempos, onde cada um equivale a execução das 1 carga só.

Além disso, na arquivo xml utilizado (*XML-PATH/simulation-COMNET.xml*), foi definida uma topologia física com 24 nós e 86 arestas com 100000 chamadas que podem requisitar de 12.5 MBps a 100 MBps, com *step* de 6.25 MBps.

4.1 Lista de Adjacências

O primeiro teste executado é a troca da estrutura de dados utilizada para representar o grafo de uma Matriz de Adjacências por uma Lista de Adjacências. Essa troca melhora a complexidade assintótica de algumas funções e piora a complexidade de outras.

Como visto nas Figuras 4.1 e 4.2, há um aumento significativo no tempo de execução do simulador com essa modificação, indicando que as funções utilizadas no simulador estão melhor implementadas para uso junto a matriz de adjacências.

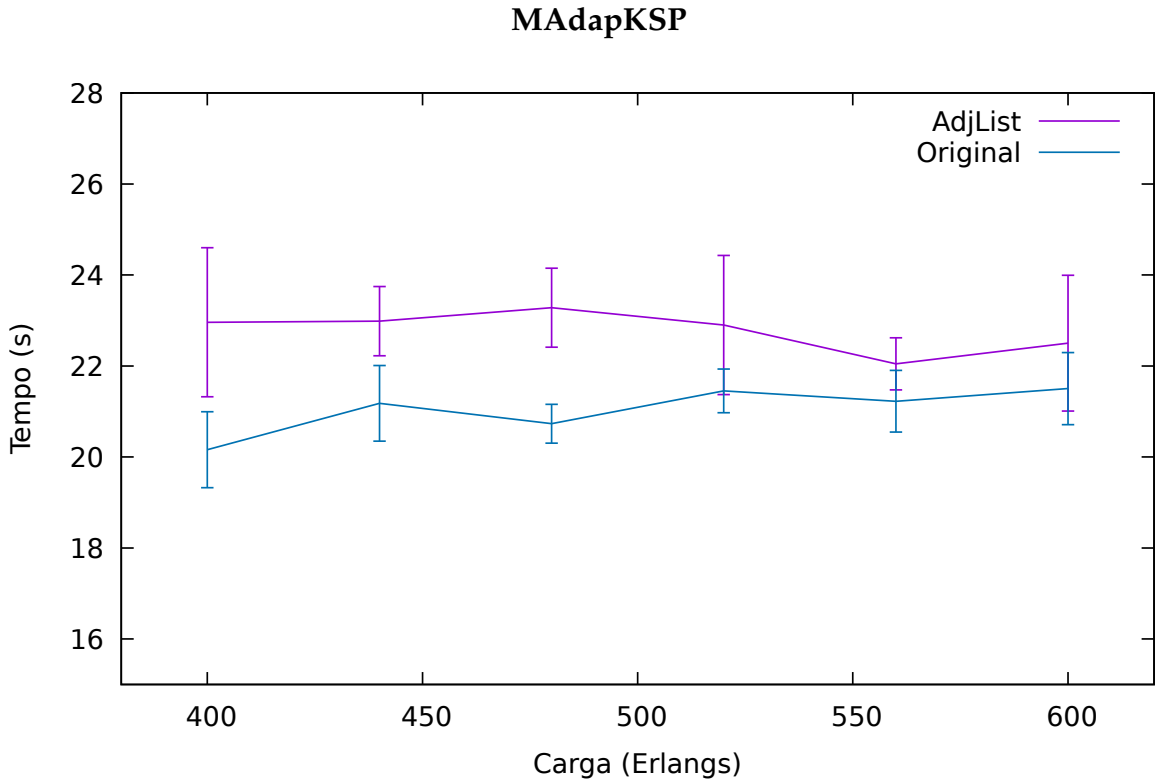

Figura 4.1: Tempo de execução do arquivo *WGA.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

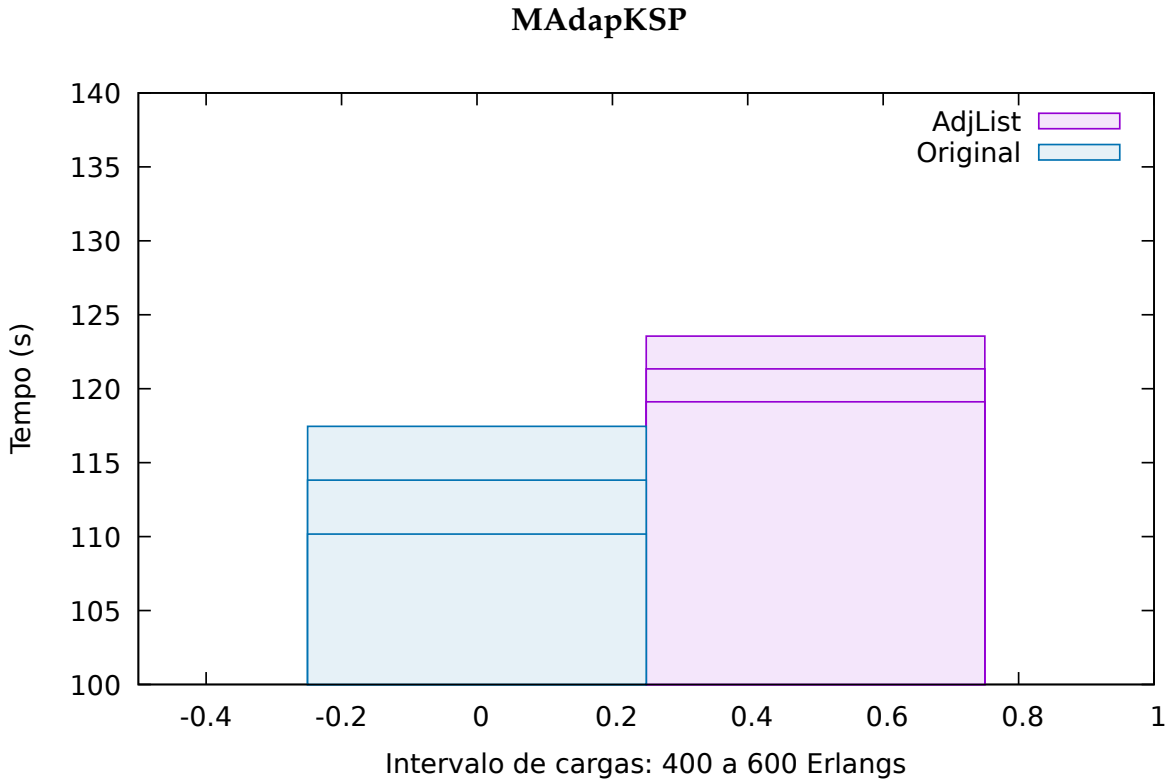

Figura 4.2: Tempo de execução do arquivo *WGA.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Já nas Figuras 4.3 e 4.4, verificou-se que em vários casos não houve uma mudança significativa do tempo de execução, porém foi aumentado significativamente o desvio padrão da execução, permitindo-se que em algumas execuções haja uma redução e em outras um aumento do tempo.

Essa imprevisibilidade no tempo de execução não é um fator desejado.

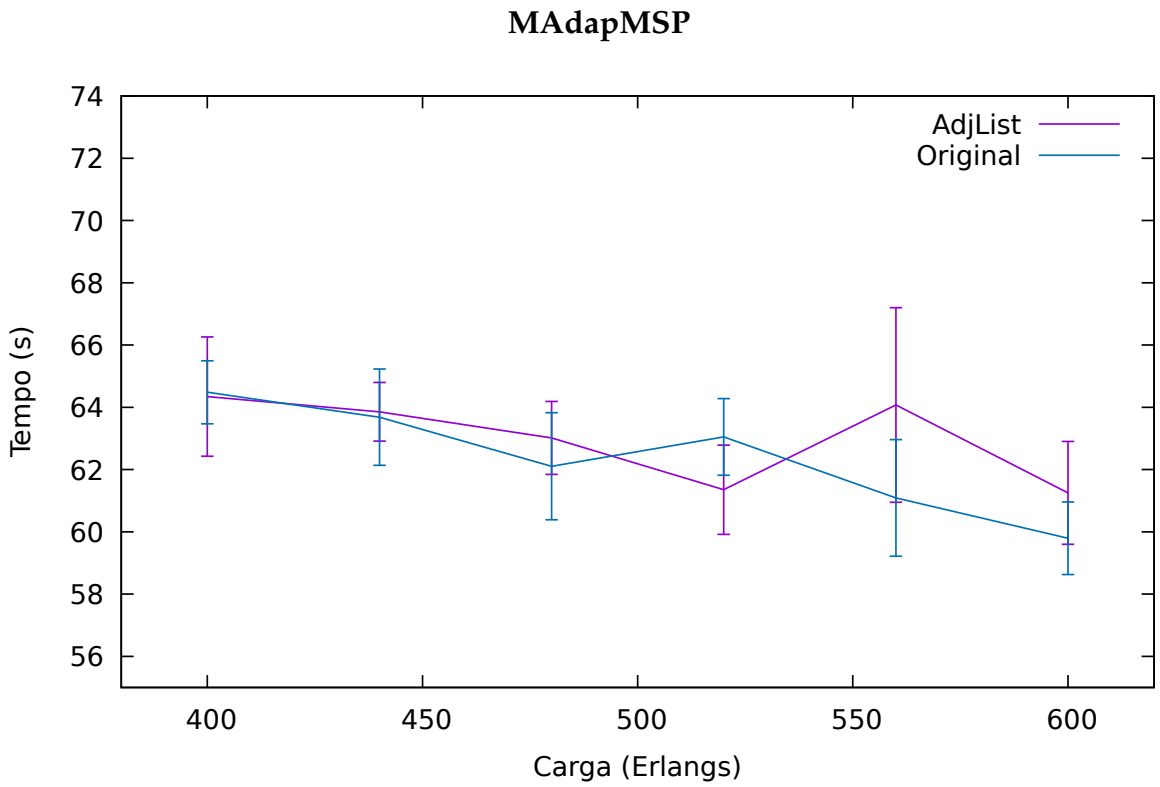

Figura 4.3: Tempo de execução do arquivo *WGA.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

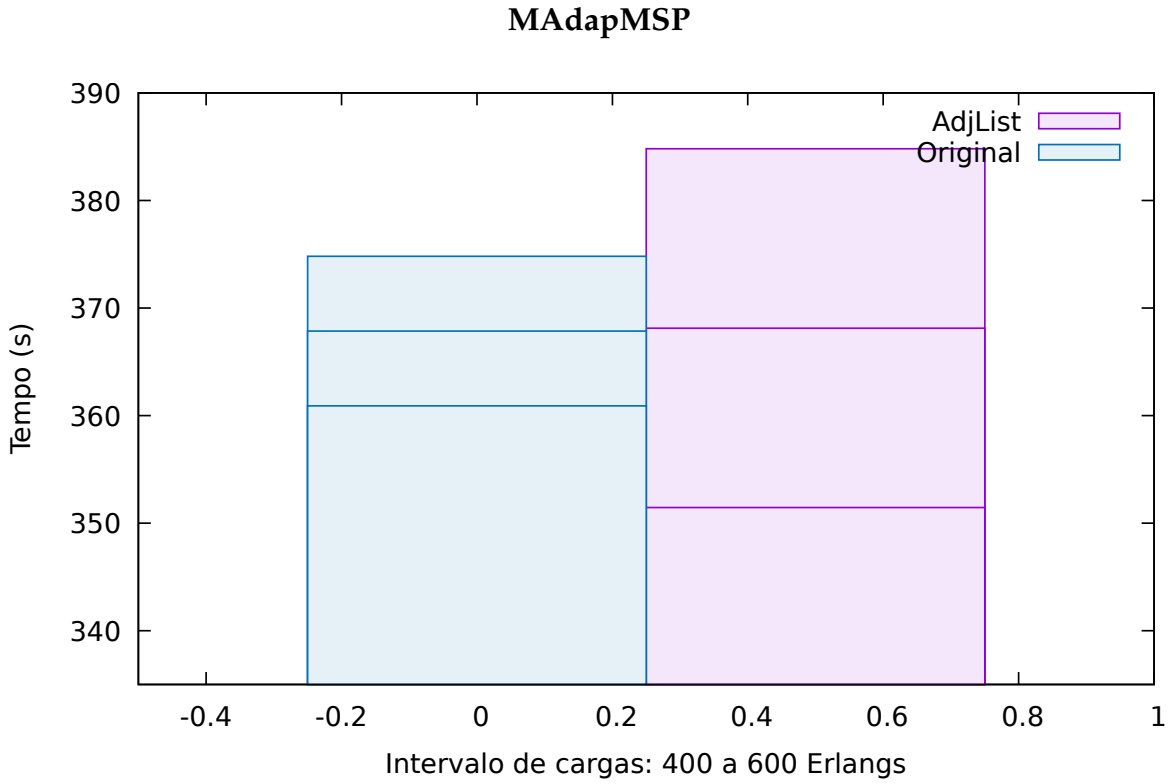

Figura 4.4: Tempo de execução do arquivo *WGA.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Devido à isso e ao aumento do tempo no outro teste, conclui-se que essa modificação não é recomendada para adição ao simulador.

4.2 Floyd Warshall

Na modificação do Floyd Warshall, encontra-se um resultado inesperado, ao comparar-se as Figuras 4.5 e 4.6, percebe-se que ao executar-se somente 1 carga no simulador, o resultado foi pior do que o simulador original, porém quando executa-se uma faixa de cargas, obtêm-se a modificação apresenta um tempo de execução mais rápido.

Perceba-se que em ambos os casos a diferença entre os tempos é pequena, dentre da faixa de aproximadamente 3.5 segundos no gráfico menor (cerca de 13.6%) e de aproximadamente 5 segundos no gráfico maior (cerca de 4.46%).

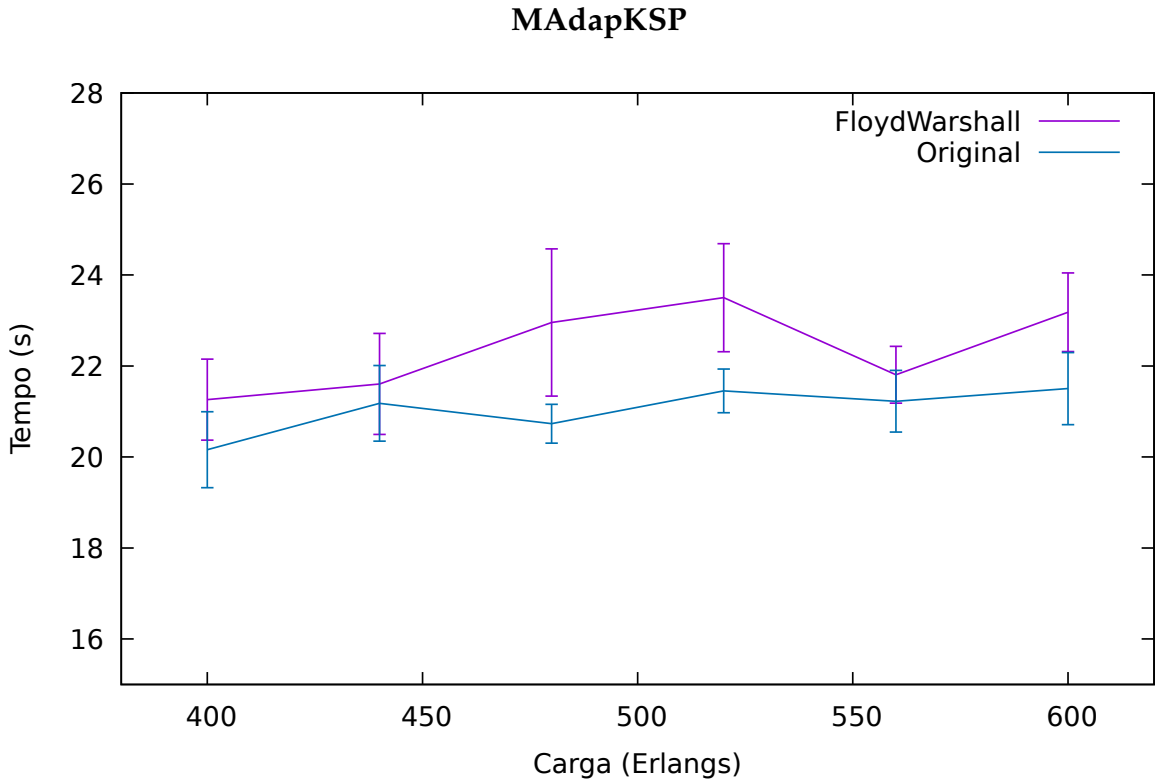

Figura 4.5: Tempo de execução do arquivo *WGFW.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

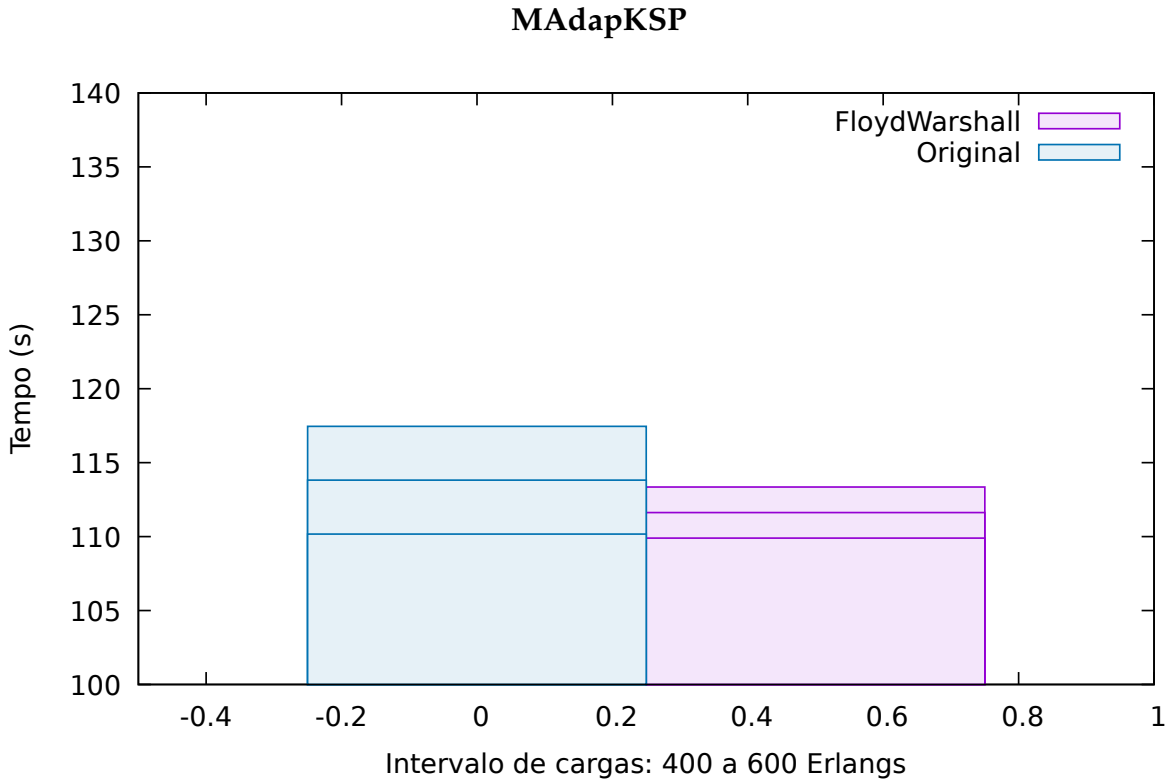

Figura 4.6: Tempo de execução do arquivo *WGFW.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Analogamente, nas Figuras 4.7 e 4.8 há uma inversão no desempenho de um gráfico para o outro e a diferença entre os tempos é de cerca de 5 segundos (7.8%) em 4.7 e cerca de 4.5 segundos (1.25%) em 4.8.

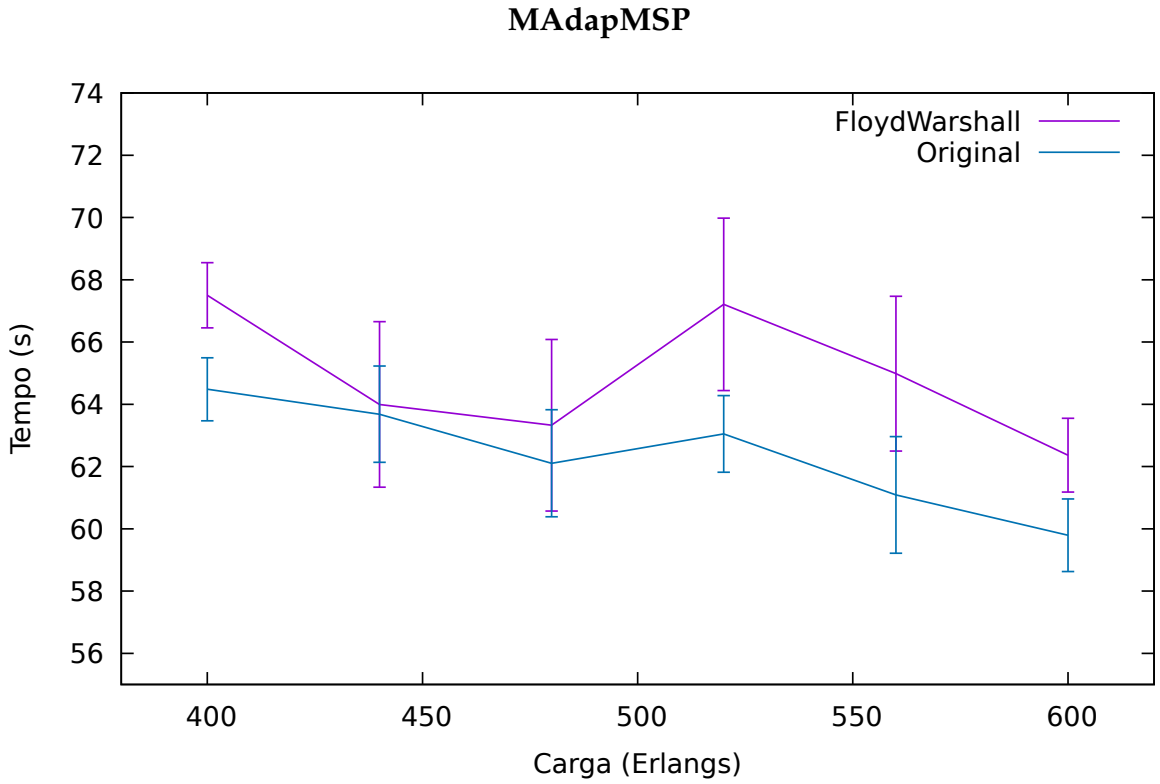

Figura 4.7: Tempo de execução do arquivo *WGFW.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

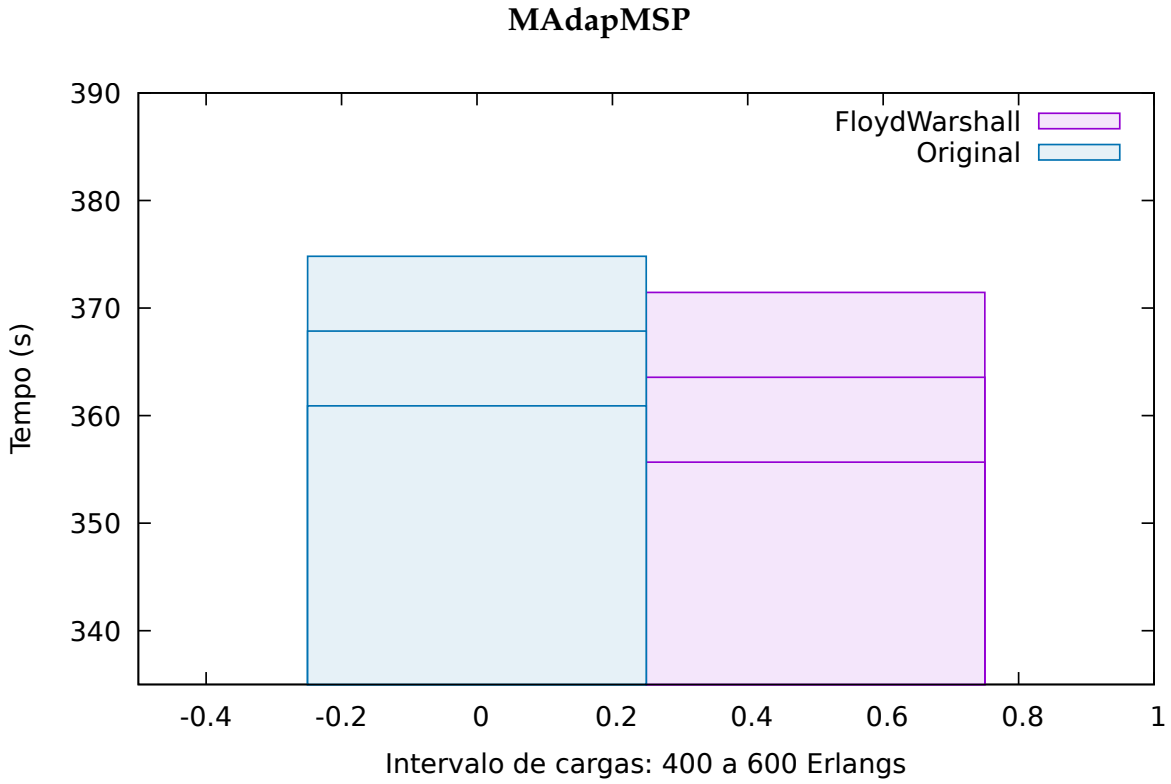

Figura 4.8: Tempo de execução do arquivo *WGFW.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Sabendo-se que nos estudos de redes ópticas elásticas é comum a simulação para várias cargas diferentes, então o uso desse algoritmo terá na maioria dos casos uma pequena melhoria na complexidade. Além disso, o algoritmo de Floyd-Warshall possui um código simples, o que facilita a sua manutenção.

Por isso, sugere-se acrescentar essa modificação no simulador.

4.3 Clustering - Multiplicação de Matrizes

A utilização de multiplicação de matrizes para as funções de clustering não apresentou grandes mudanças nas Figuras 4.9, 4.10 e 4.11, estando no intervalo de confiança do simulador original.

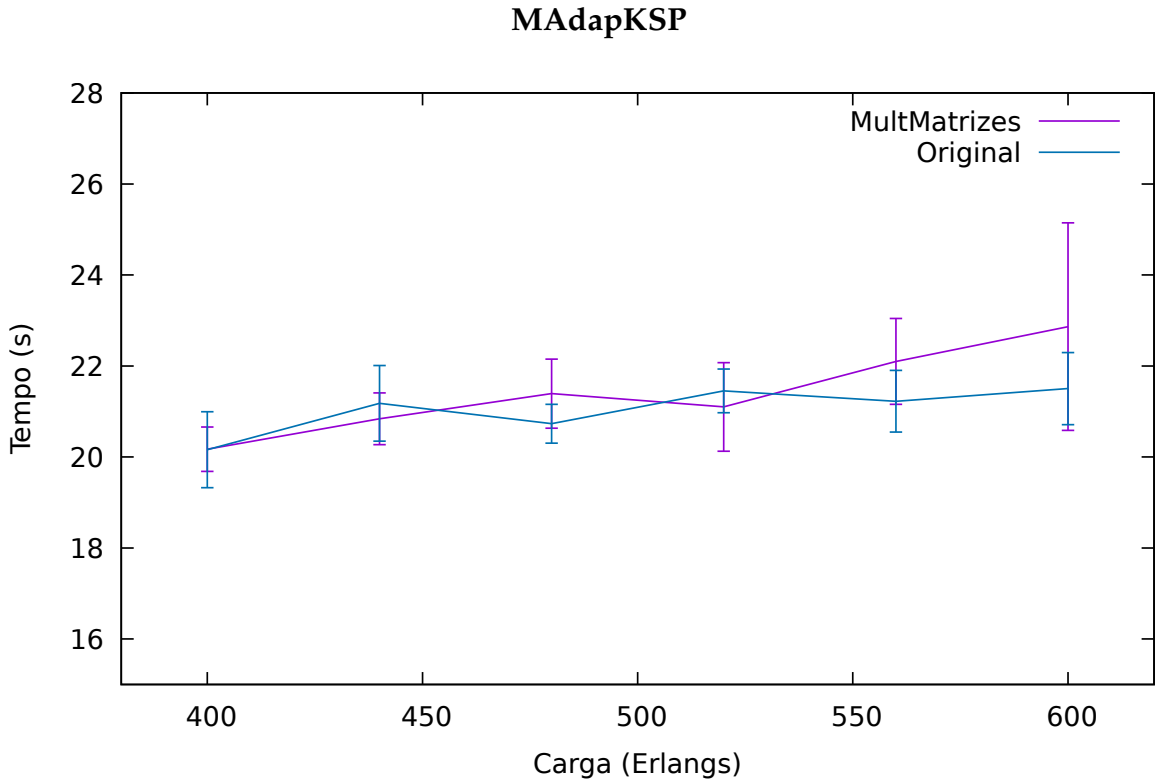

Figura 4.9: Tempo de execução do arquivo *WGC.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

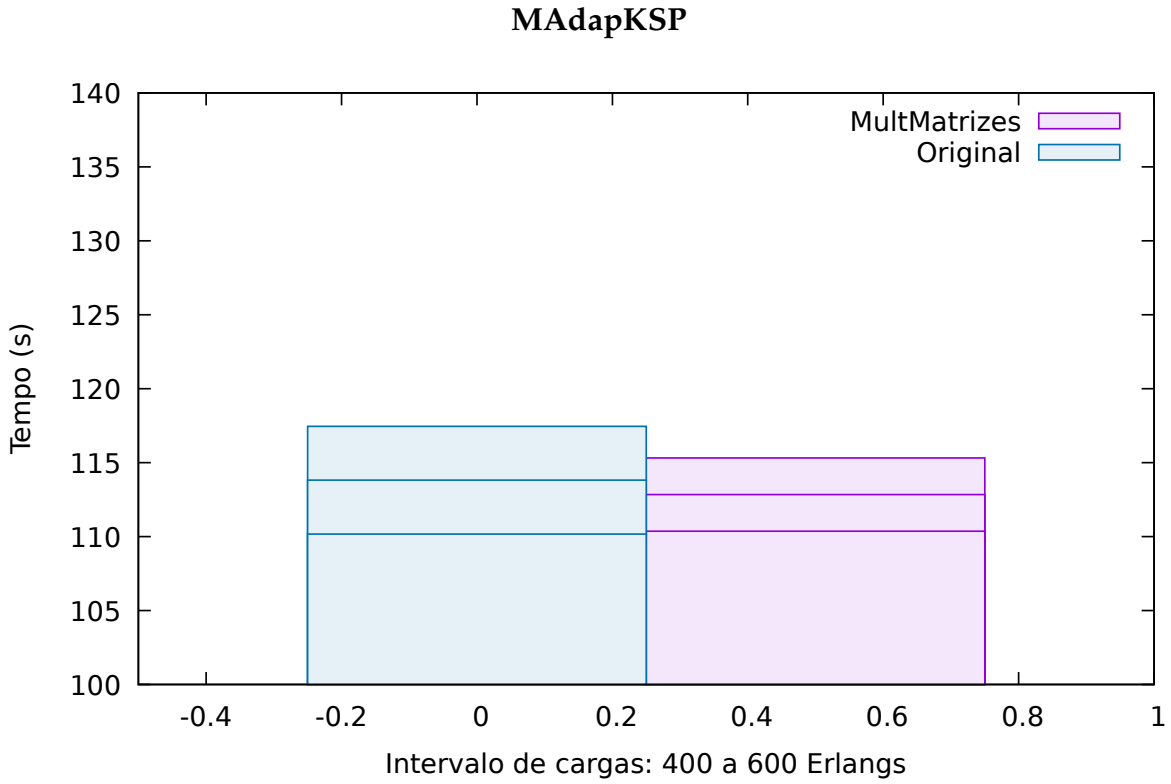

Figura 4.10: Tempo de execução do arquivo *WGC.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

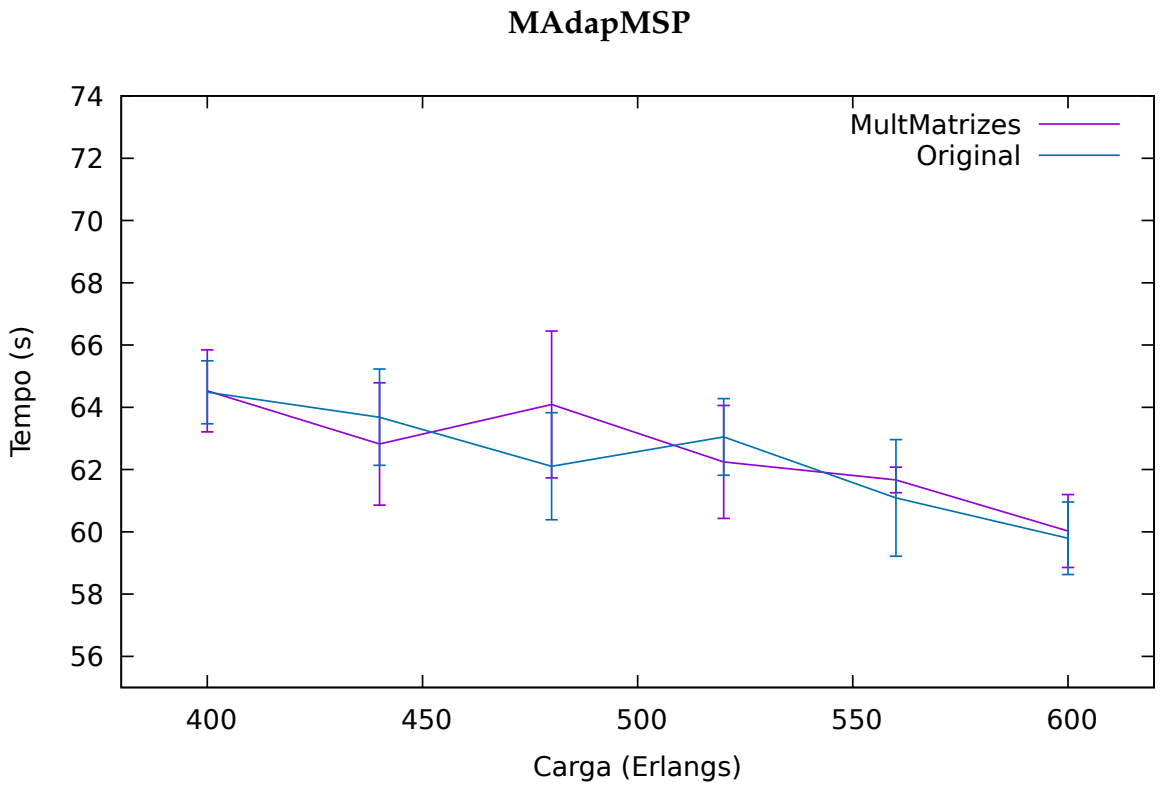

Figura 4.11: Tempo de execução do arquivo *WGC.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

Percebe-se porém uma melhoria de cerca de 15 segundos (4.16%) no teste 4.12

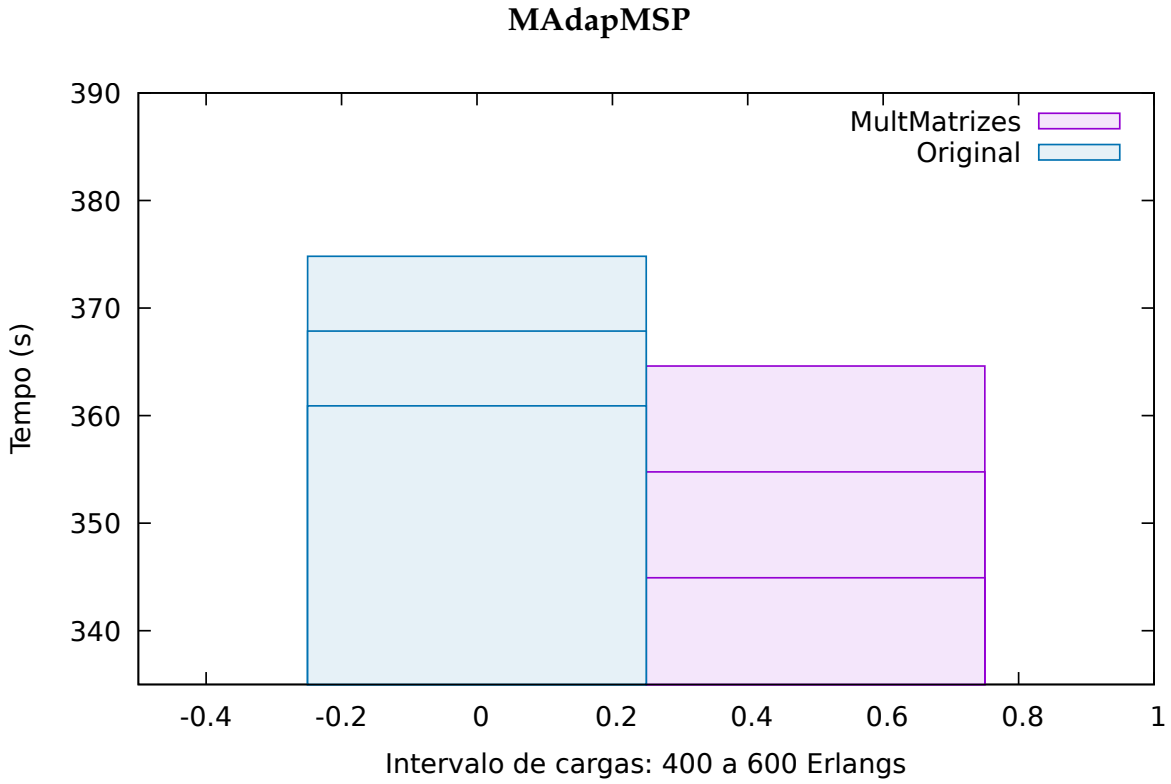

Figura 4.12: Tempo de execução do arquivo *WGC.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Logo, analogamente ao algoritmo de Floyd-Warshall, essa modificação é recomendada para inclusão no simulador devido a sua simplicidade e melhorias pequenas no tempo de execução.

4.4 Dijkstra - Min Heap

Uma da modificações responsáveis por influenciar a proposta desse trabalho foi o algoritmo clássico de Dijkstra, conhecido por ser um algoritmo que pode ter a complexidade melhorada através do uso de heap.

Contrário ao que se podia esperar, apesar de uma melhora na complexidade assintótica, ao executar-se o algoritmo, percebeu-se um aumento significativo no tempo de execução de cerca de 20% e de 7.02% em relação à implementação original do simulador, Figuras 4.13 e 4.14.

Esse comportamento pode ser visto pelo fato de que nas simulações reais utilizadas, o número de nós e arestas no grafo são pequenos. Assim, apesar de haver uma melhora

teórica, na prática a quantidade de nós não é grande o suficiente para que essa melhora ocorra.

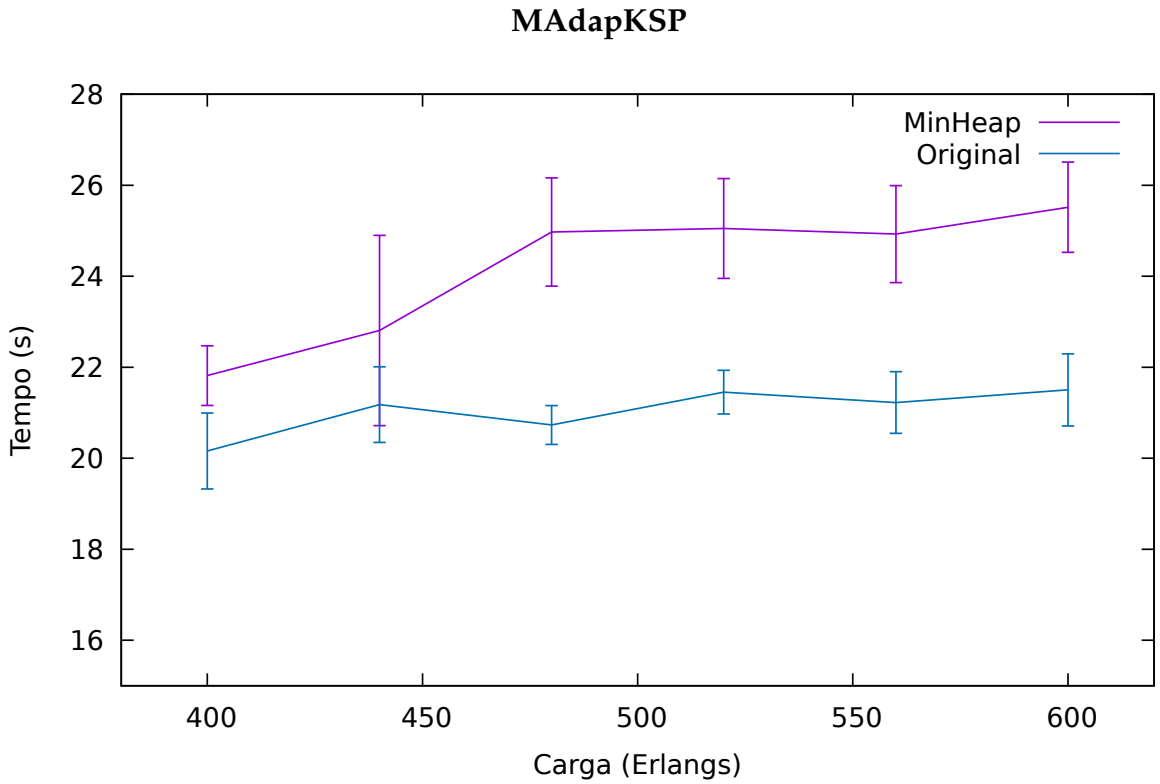

Figura 4.13: Tempo de execução do arquivo *DM.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

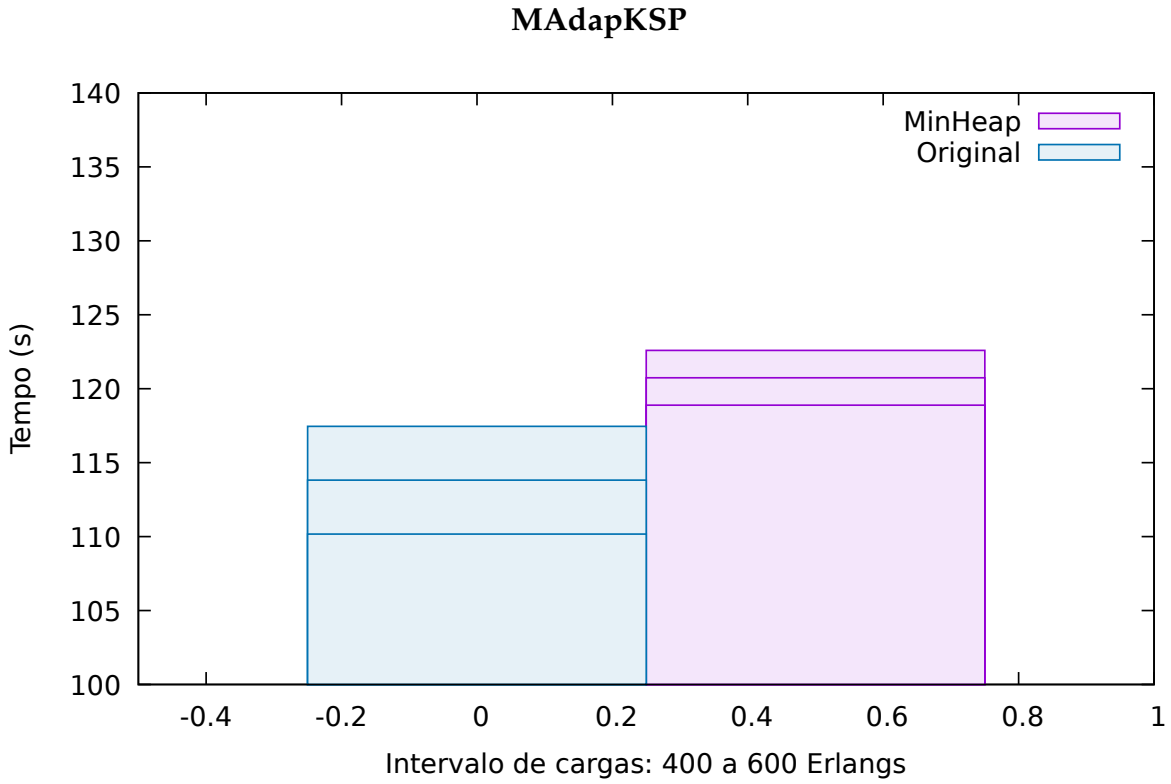

Figura 4.14: Tempo de execução do arquivo *DM.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Apesar disso, a estrutura de dados Min Heap não apresentou o mesmo comportamento no algoritmo RA *MAdapMSP*, no qual o tempo de execução foi virtualmente o mesmo do simulador original.

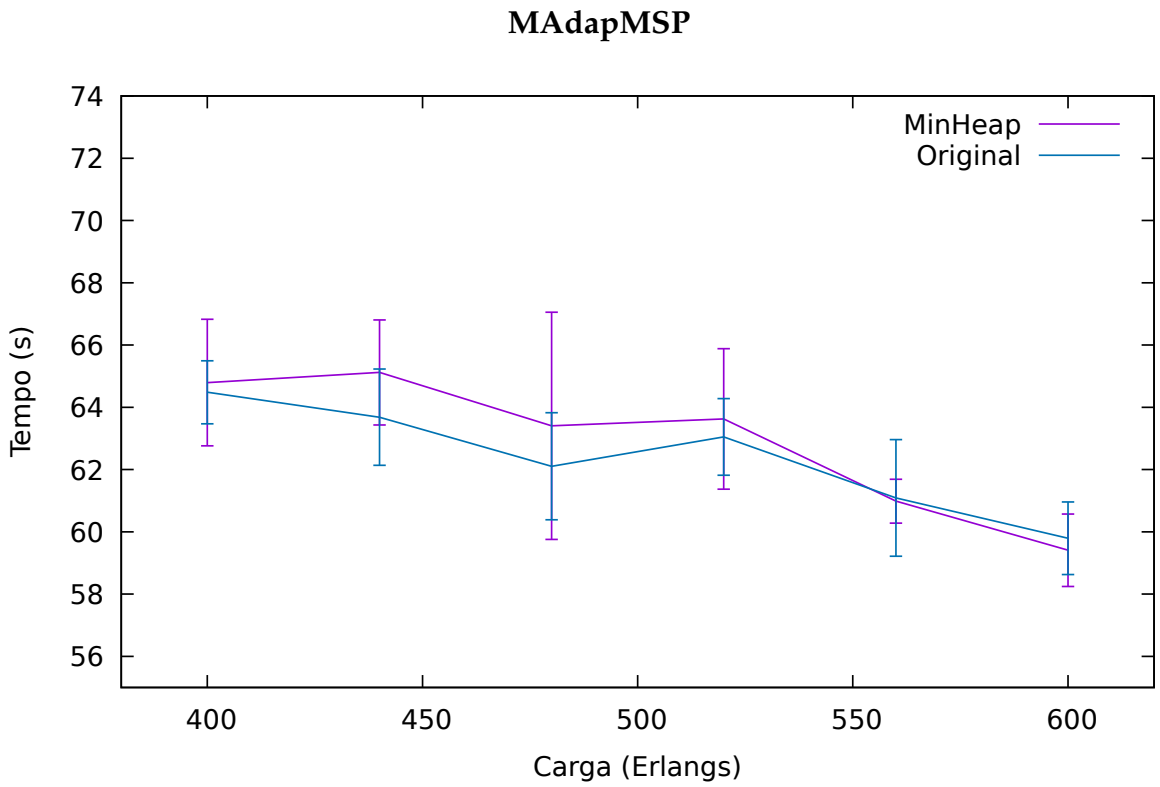

Figura 4.15: Tempo de execução do arquivo *DM.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

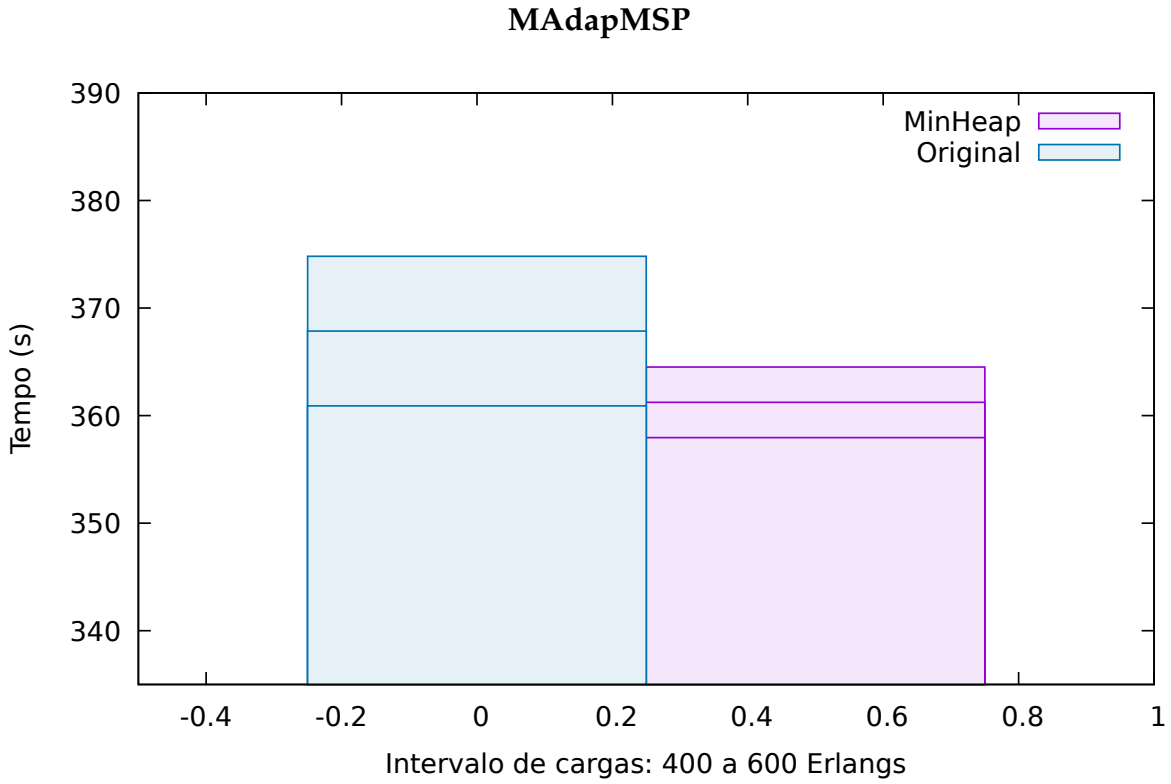

Figura 4.16: Tempo de execução do arquivo *DM.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Como essa modificação aumenta significativamente a complexidade do simulador sem haver melhora no tempo de execução, piorando em alguns casos, sugere-se não acrescentá-la ao simulador.

4.5 Djisktra - Min Heap e Lista de Adjacências

Seguindo da seção anterior, o algoritmo de Dijkstra otimizado normalmente é implementado com Heap e com Lista de Adjacências ao mesmo tempo.

Como já visto tanto a lista de adjacências como o uso do Min Heap, não apresentaram bons resultados. Similarmente, percebe-se nessa seção que a utilização simultânea de ambos apresentou uma piora maior dos resultados comparando ao uso de um só.

A exceção à essa regra está na Figura 4.17, onde houve uma leve melhora em comparação à Figura 4.13.

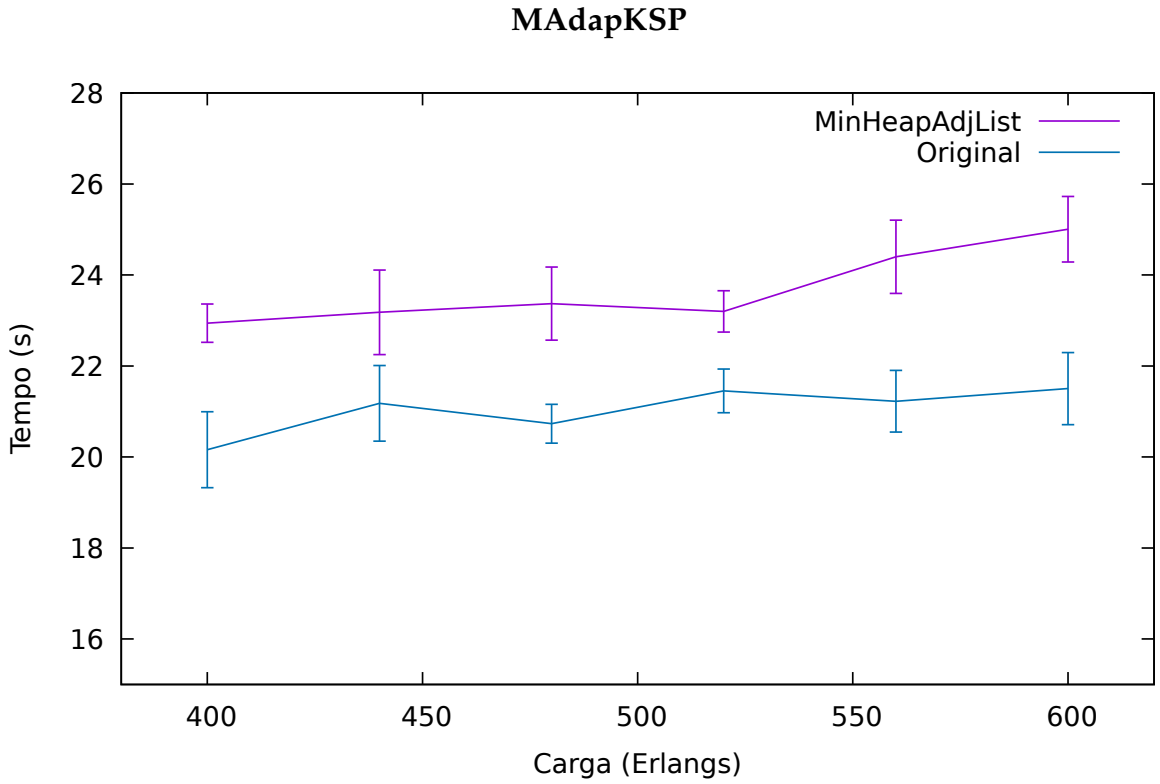

Figura 4.17: Tempo de execução do arquivo *DMA.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

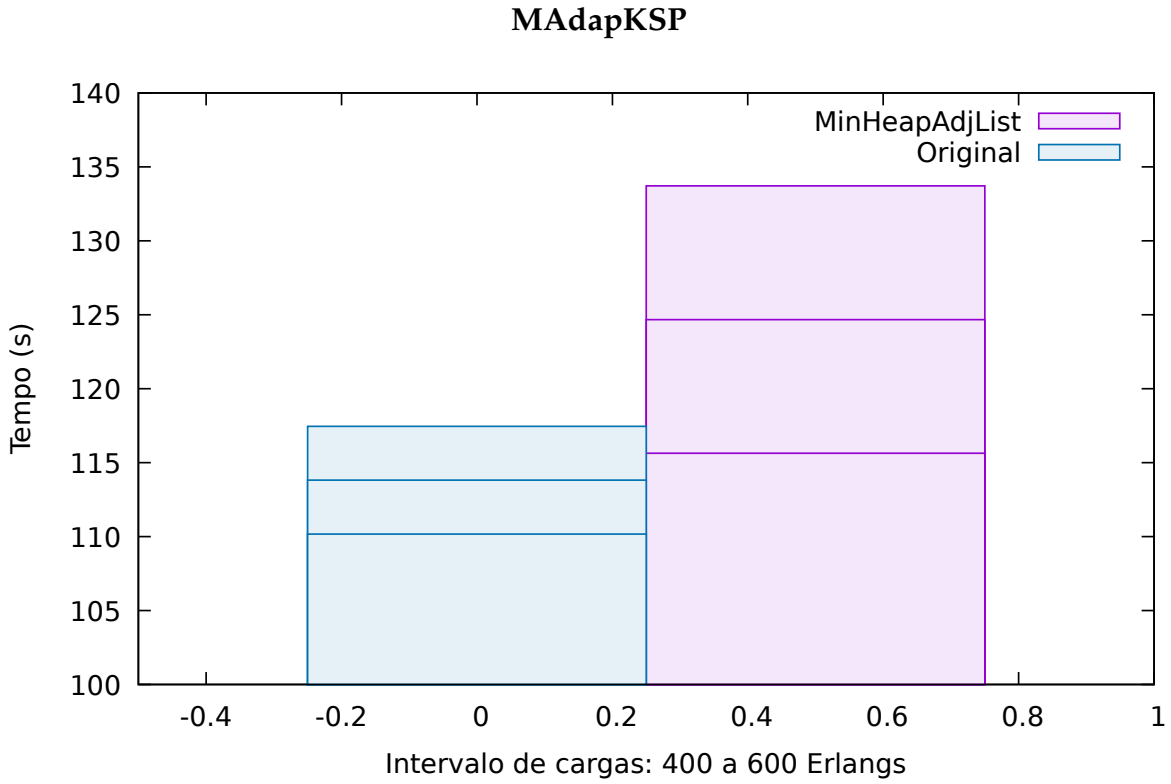

Figura 4.18: Tempo de execução do arquivo *DMA.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

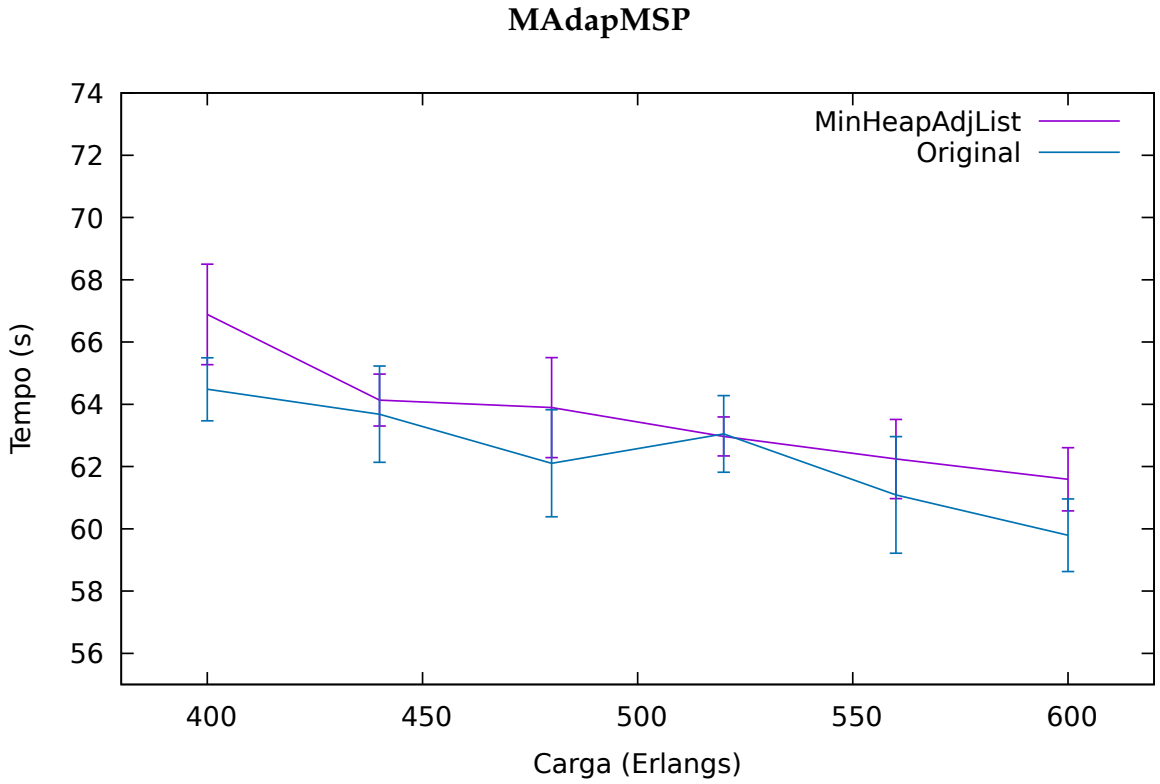

Figura 4.19: Tempo de execução do arquivo *DMA.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

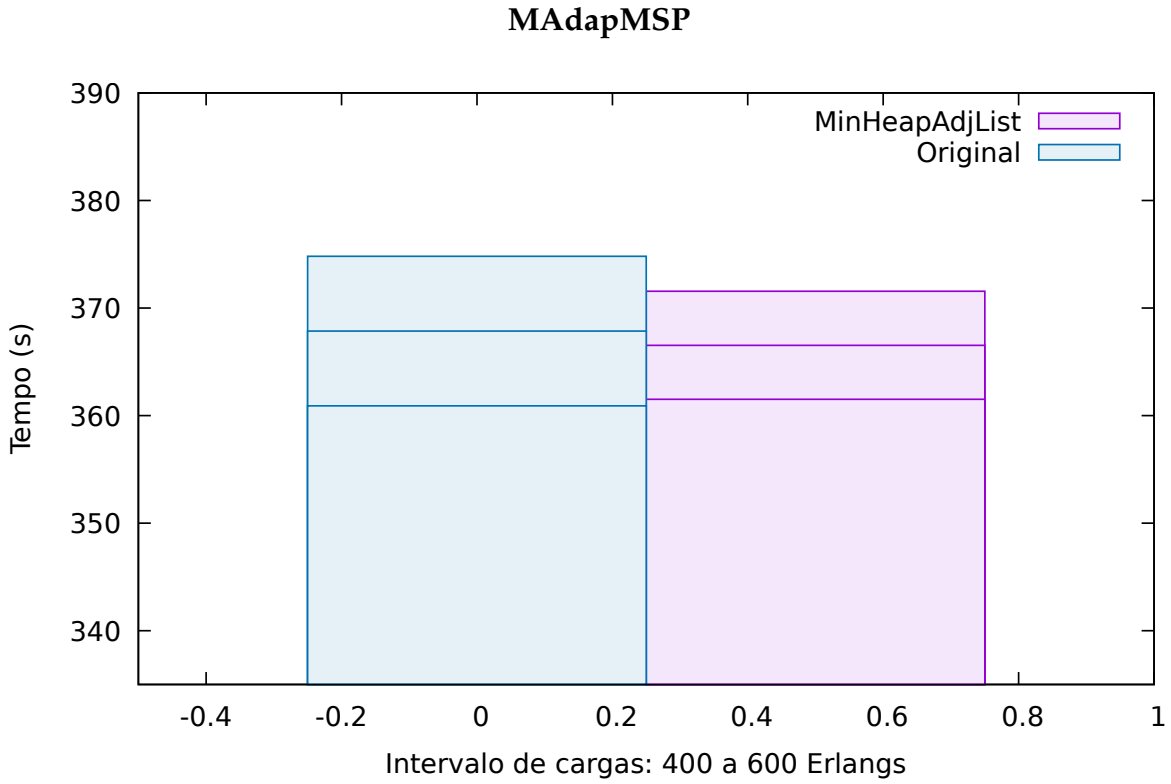

Figura 4.20: Tempo de execução do arquivo *DMA.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Ao final, conclui-se que essa modificação não apresentou melhoras e portanto não recomenda-se sua inclusão no simulador.

4.6 Djikstra - Fibonacci Heap

Analogamente ao uso do Min Heap, pelo fato de que nas simulações reais utilizadas, o número de nós e arestas no grafo são pequenos, apesar de haver uma melhora teórica, na prática a quantidade de nós não é grande o suficiente para que essa melhora ocorra, o que é percebido no desempenho das Figuras 4.21 e 4.22.

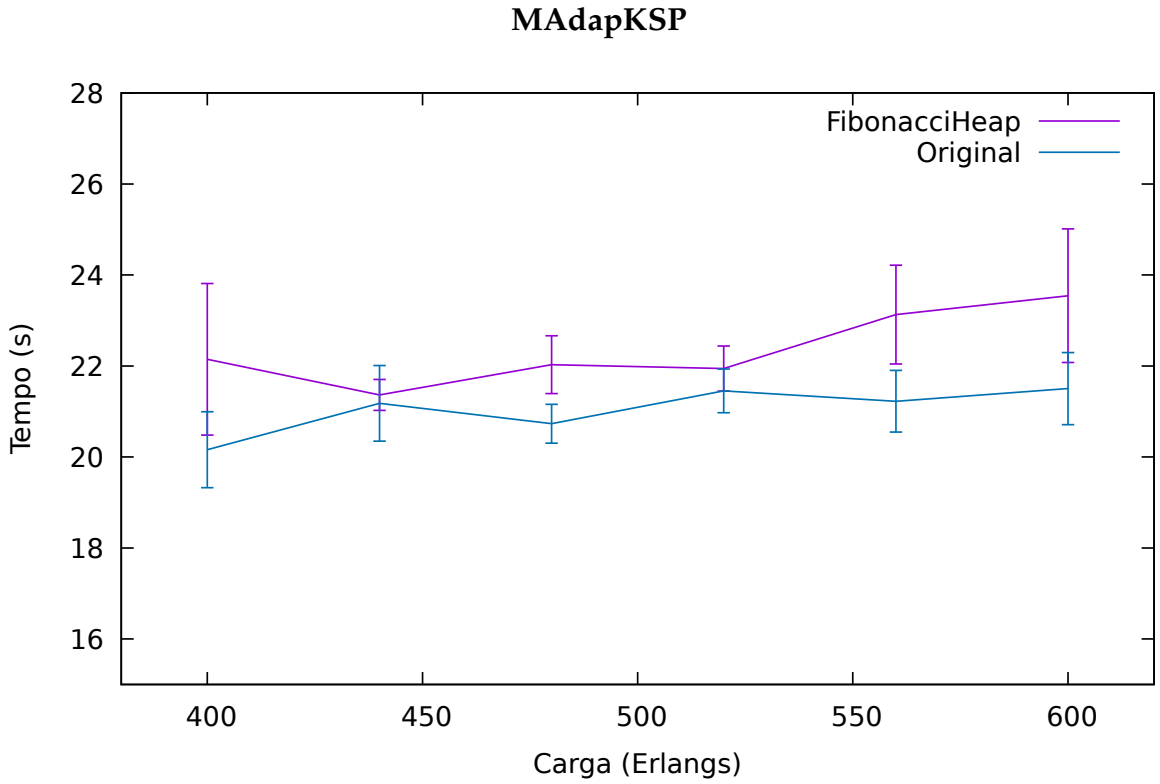

Figura 4.21: Tempo de execução do arquivo *DF.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

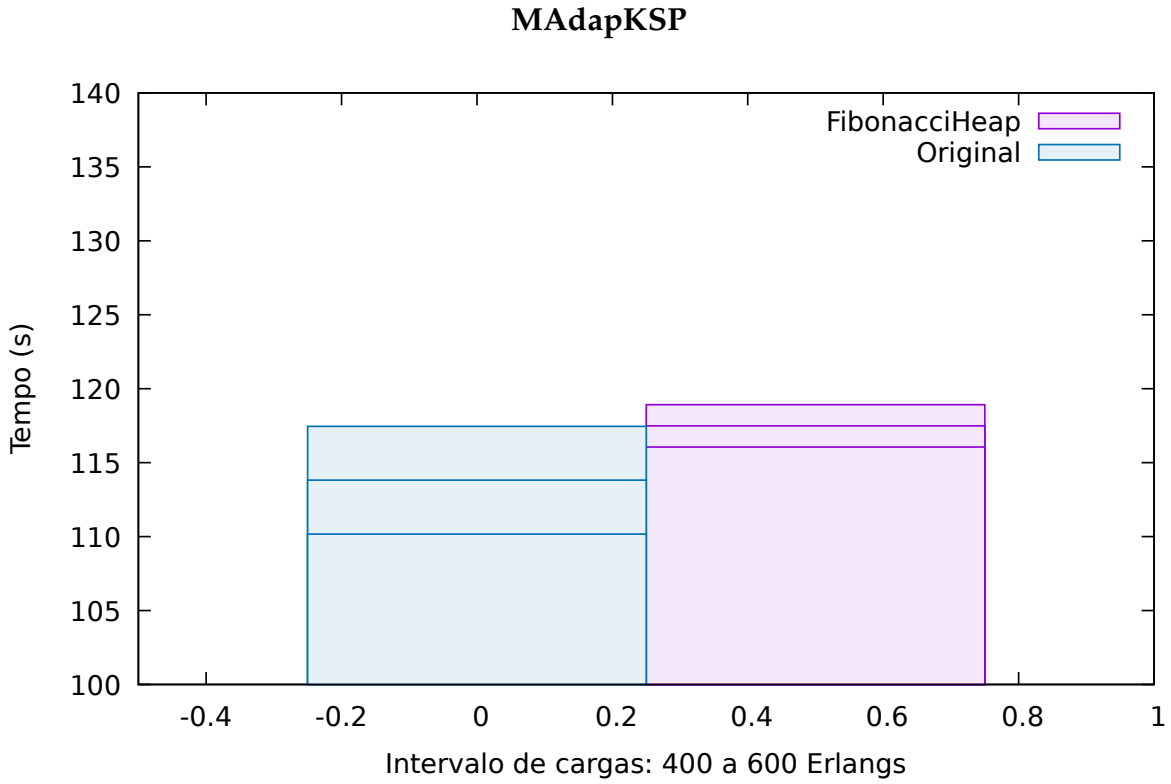

Figura 4.22: Tempo de execução do arquivo *DF.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Similarmente, no teste com maior tempo de execução *MAdapMSP*, o algoritmo não apresentou uma diferença significativa no tempo de execução.

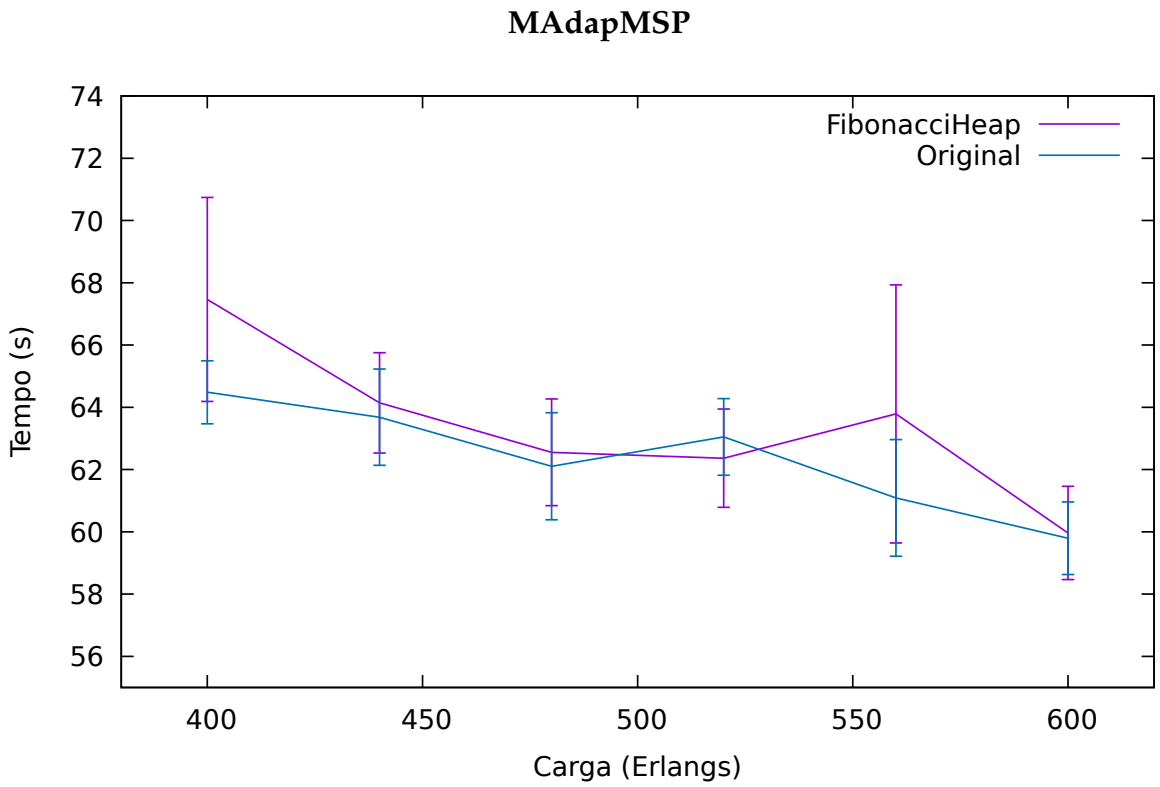

Figura 4.23: Tempo de execução do arquivo *DF.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

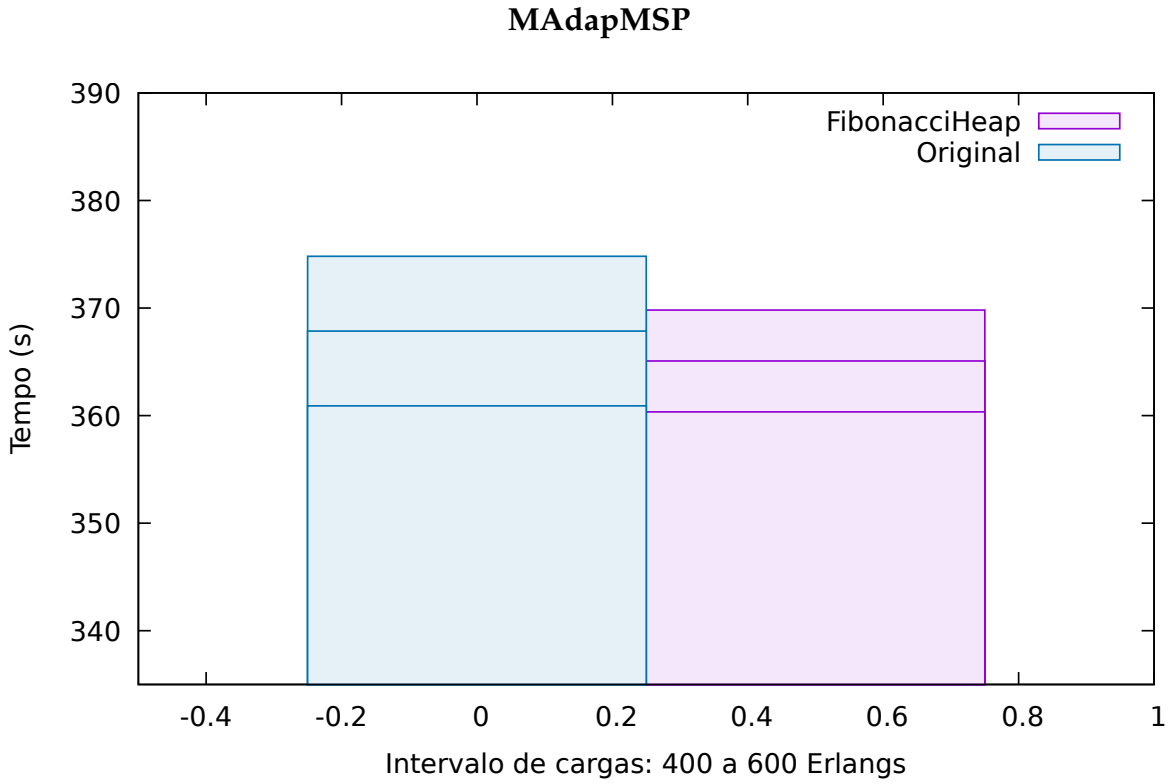

Figura 4.24: Tempo de execução do arquivo *DF.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Para concluir esse algoritmo, o resultado inicial esperado, uma melhoria no tempo proveniente da complexidade assintótica, não foi obtido e, logo, essa modificação não é recomendada para adoção no simulador.

Nota-se porém que também será considerado o uso desse simulador junto a lista de adjacências.

4.7 Djisktra - Fibonacci Heap e Lista de Adjacências

A última das modificações relacionada ao problema do menor caminho é o uso simultâneo da Fibonacci Heap e da Lista de Adjacências.

No algoritmo *MAdapKSP*, o resultado obtido está de acordo com o já observado, houve um aumento de até 15% no tempo de execução do algoritmo.

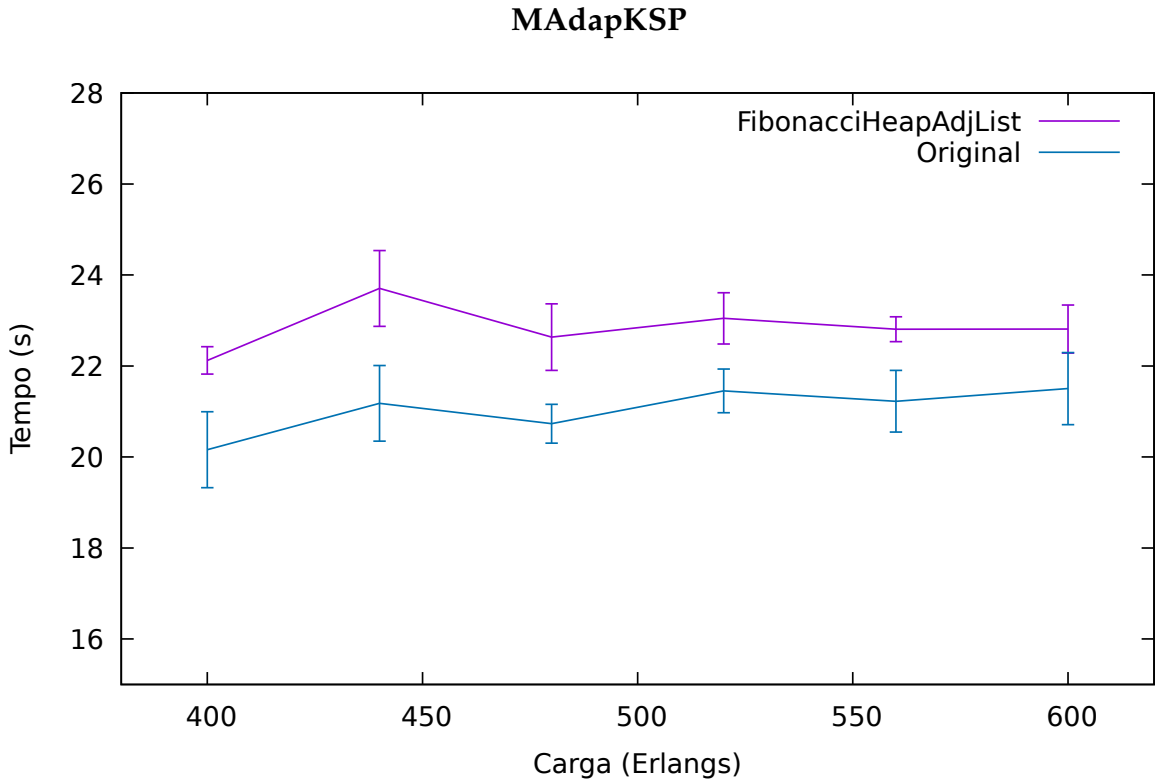

Figura 4.25: Tempo de execução do arquivo *DFA.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

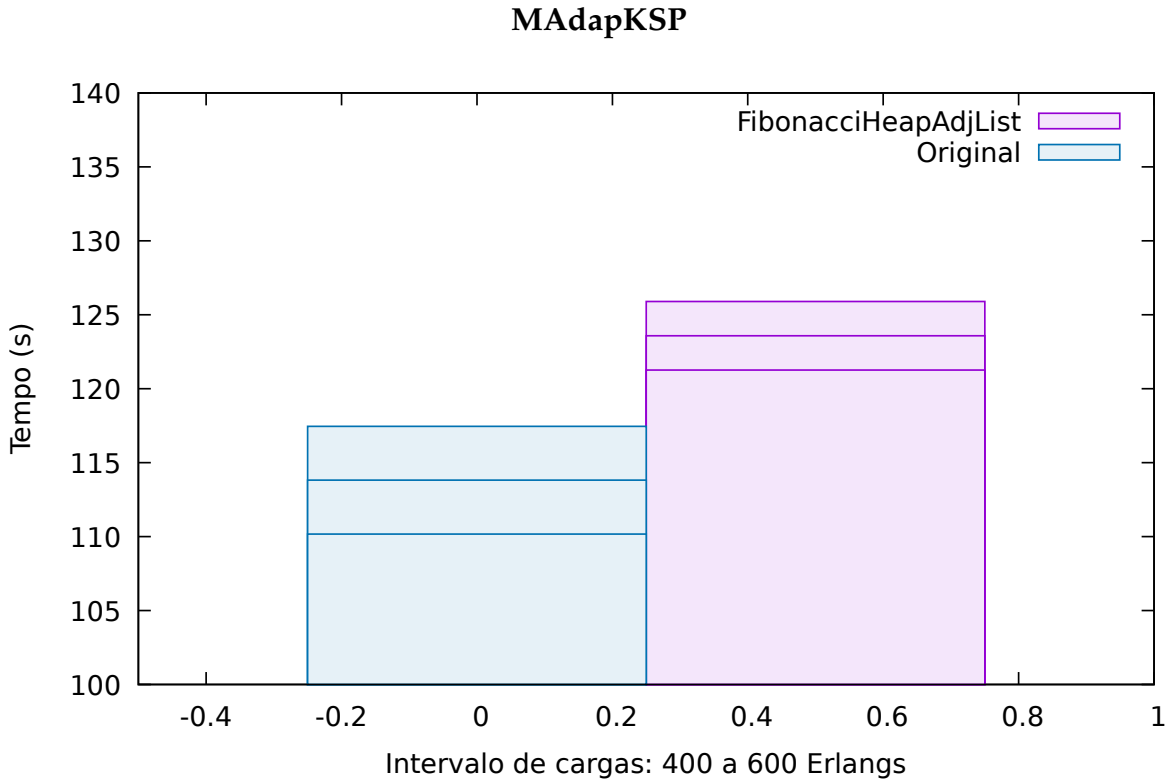

Figura 4.26: Tempo de execução do arquivo *DFA.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

No entanto, para a execução mais longa, obteve-se um resultado positivo. Na Figura 4.27, não há grandes diferenças no tempo de execução, porém na Figura 4.28 há uma melhora de cerca de 11 segundos (3,07%) no resultado.

Vale-se mencionar que essa mudança não se deve a um aumento no número de nós da topologia física, uma vez que ela é a mesma para ambos os algoritmos.

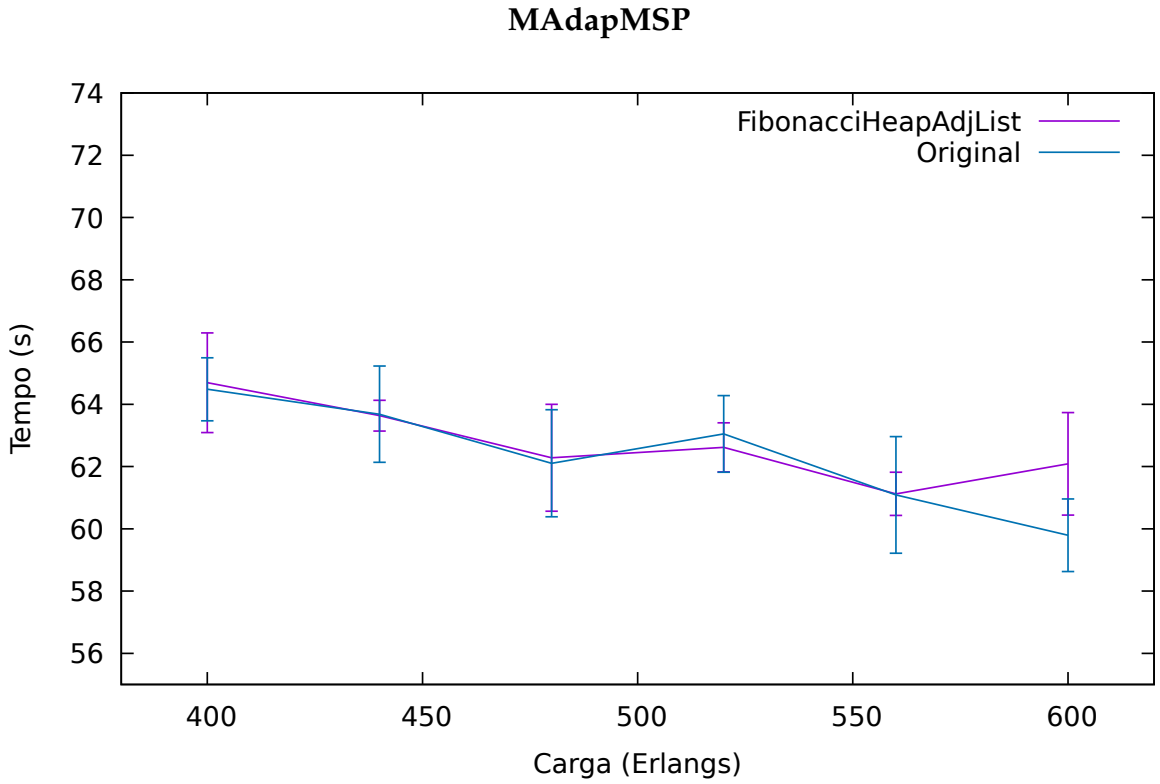

Figura 4.27: Tempo de execução do arquivo *DFA.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

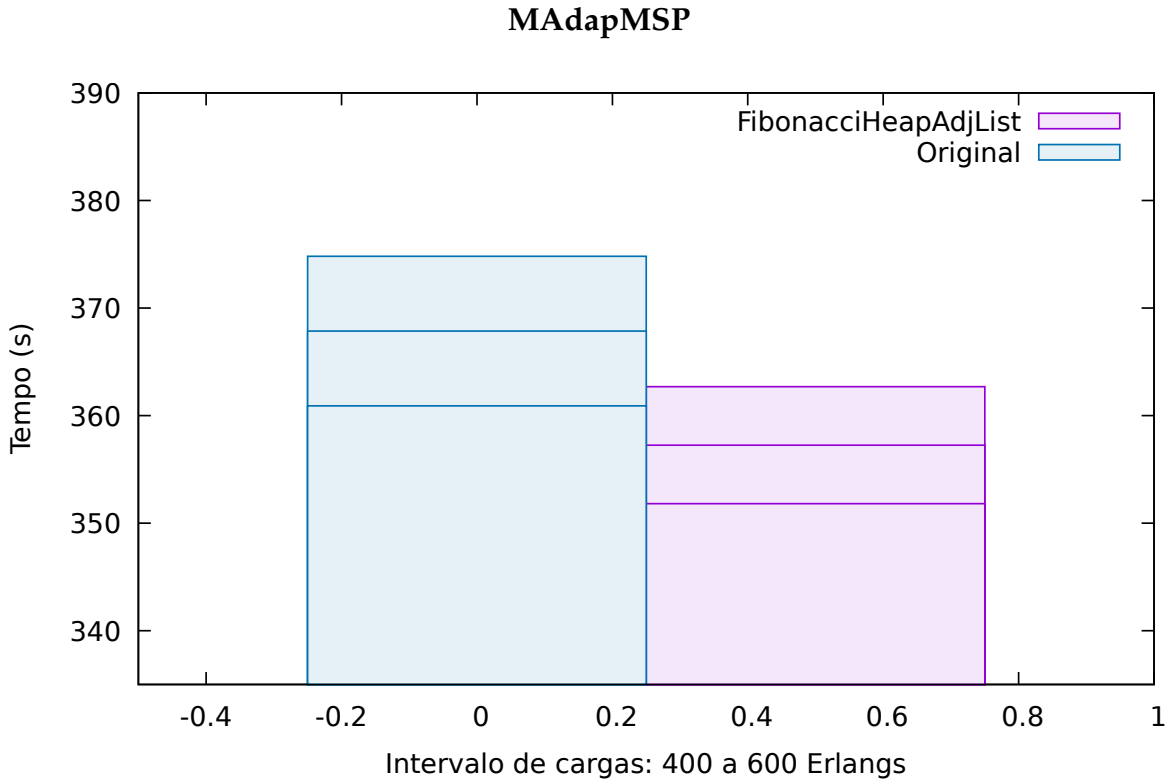

Figura 4.28: Tempo de execução do arquivo *DFA.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Essa diferença torna difícil a tomada de decisão em relação à esse algoritmo, uma vez que o tempo de execução é considerado mais importante quando possui-se execuções mais demoradas e logo, faz sentido a utilização do algoritmo que seja melhor para o *MAdapMSP*. Porém, a diferença é percentualmente pequena e há um aumento significativo na manutenção desse código, uma vez que a estrutura de dados da Fibonacci Heap é altamente complexa.

Por motivos de manutenção, conclui-se que o resultado obtido desse algoritmo não é positivo o suficiente para justificar sua utilização e portanto não recomenda-se sua utilização no simulador.

4.8 EONLink - Resizable-Array

A modificação da função getSlotsAvailable foi sugerida através do profiling do código, sendo assim, ela possui a maior redução no tempo de execução com a algoritmo *MAdapKSP*, de até 20 % na execução de uma carga única e de cerca de 10 % na execução de todas as cargas, como visto nas Figuras 4.29 e 4.30

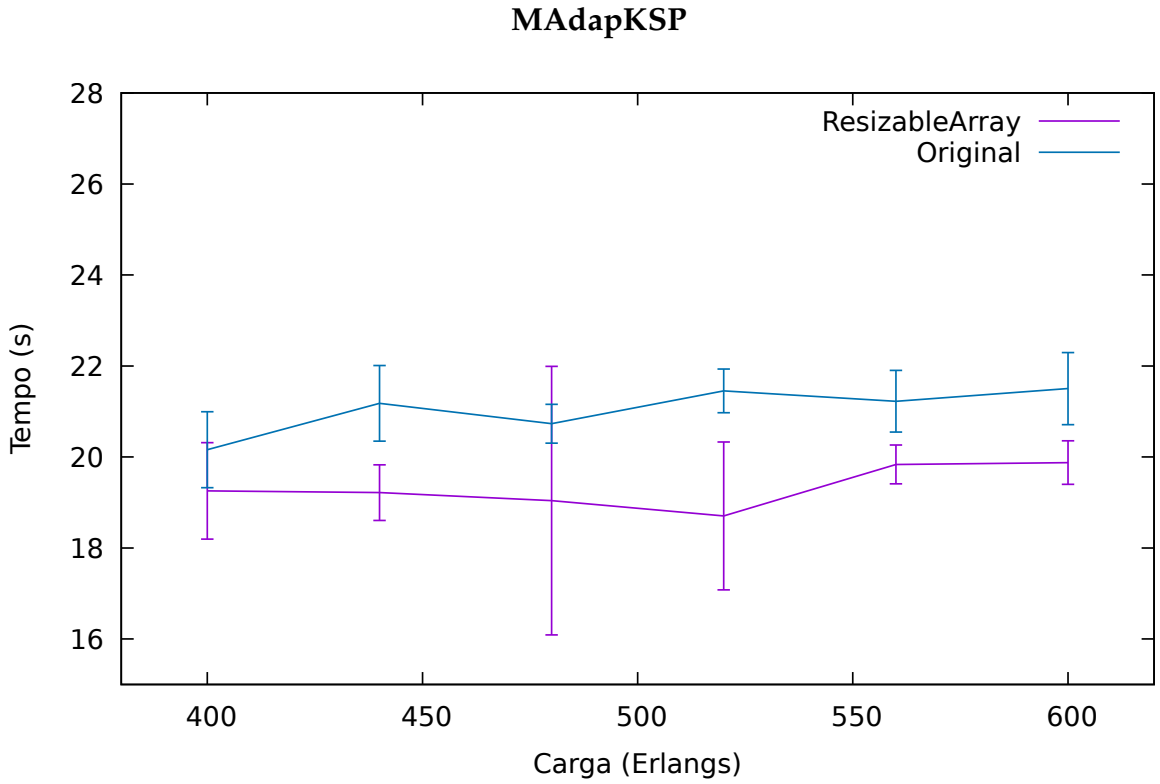

Figura 4.29: Tempo de execução do arquivo *EG.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

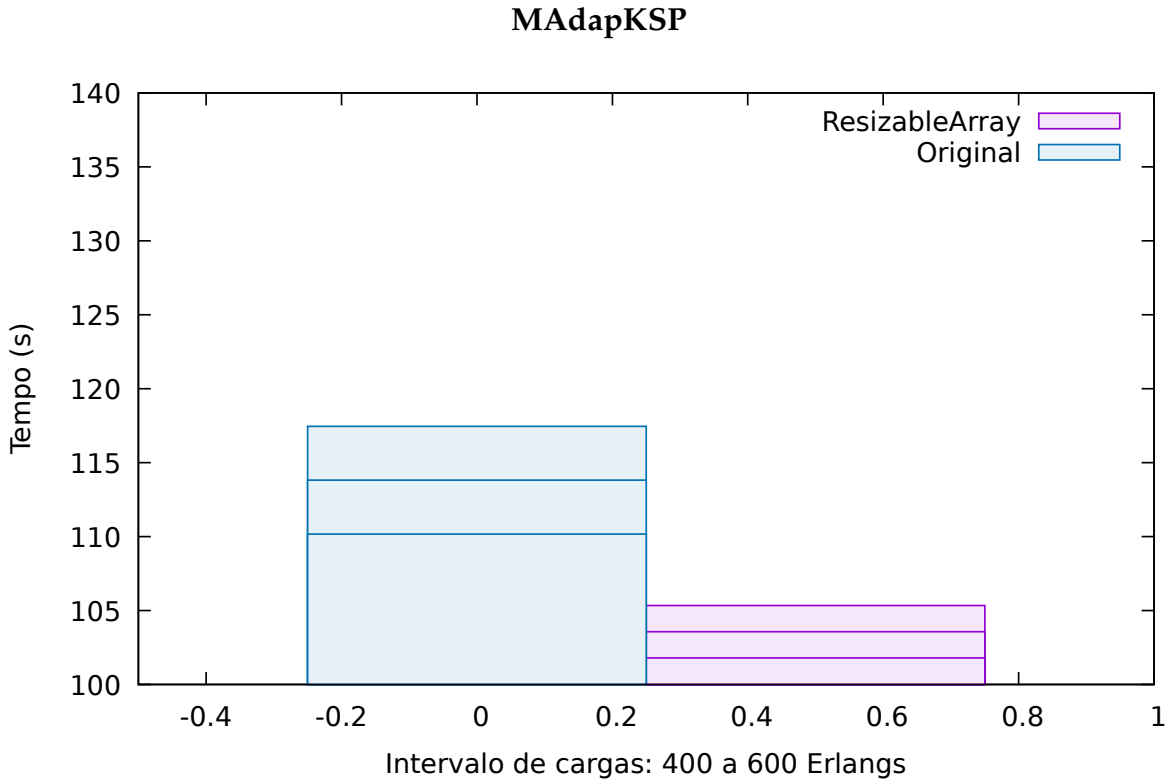

Figura 4.30: Tempo de execução do arquivo *EG.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Já nas execuções mais longas, percebe-se que não há uma grande diferença na execução entre essa modificação e o algoritmo original (Figuras 4.31 e 4.32).

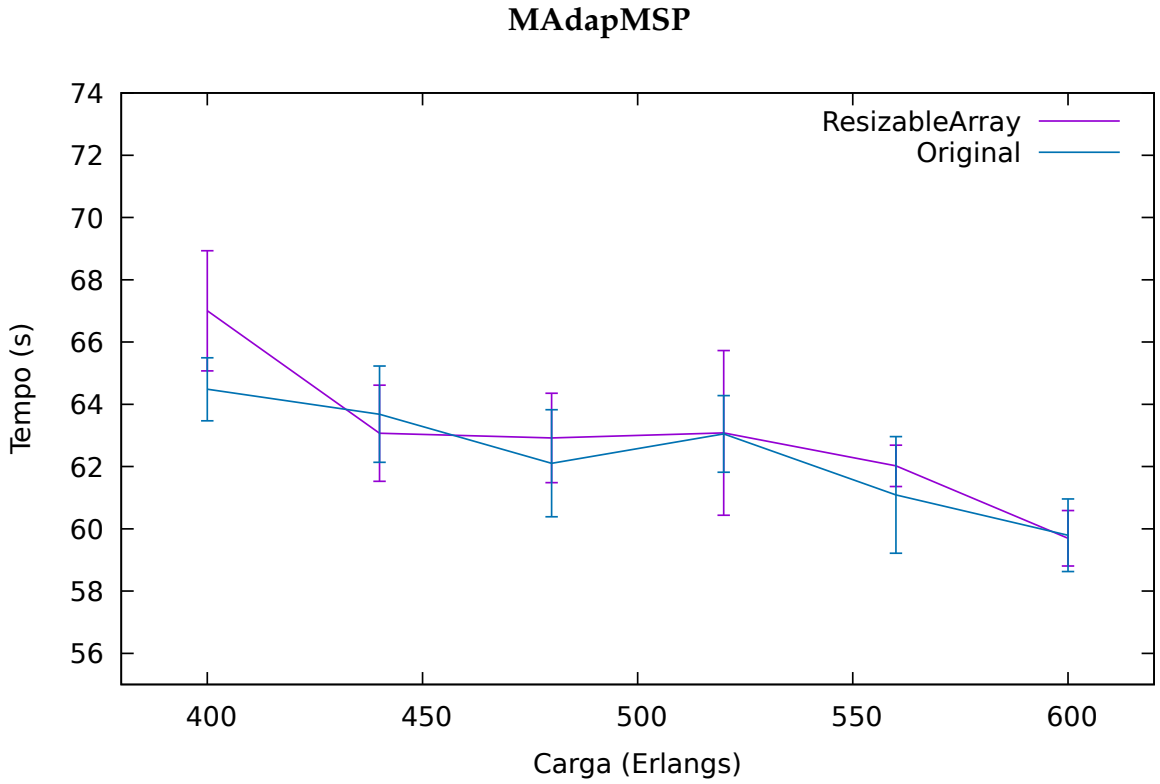

Figura 4.31: Tempo de execução do arquivo *EG.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

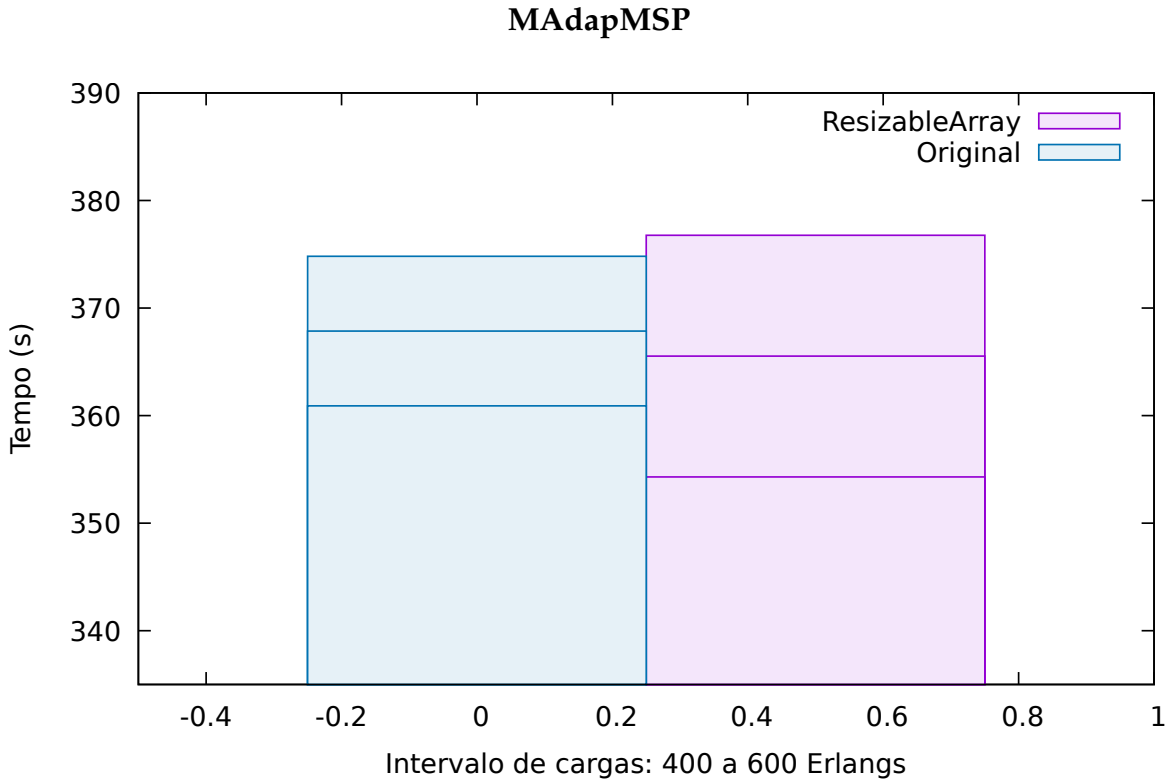

Figura 4.32: Tempo de execução do arquivo *EG.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Conclui-se que a melhoria da função getSlotsAvailable apresentou resultados bem satisfatórios que serão implementados no simulador, porém essa melhoria é menos efetiva para algumas execuções (Figuras 4.31 e 4.32).

4.9 EONLink - Greedy Approach

A outra melhoria feita a partir do profiling possui uma complexidade assintótica ainda melhor que a modificação já proposta na Seção 4.8.

No entanto, percebe-se na Figura 4.33 uma piora em relação ao simulador original. Até na Figura 4.34 onde há uma melhoria, essa é inferior a melhoria existente na seção anterior.

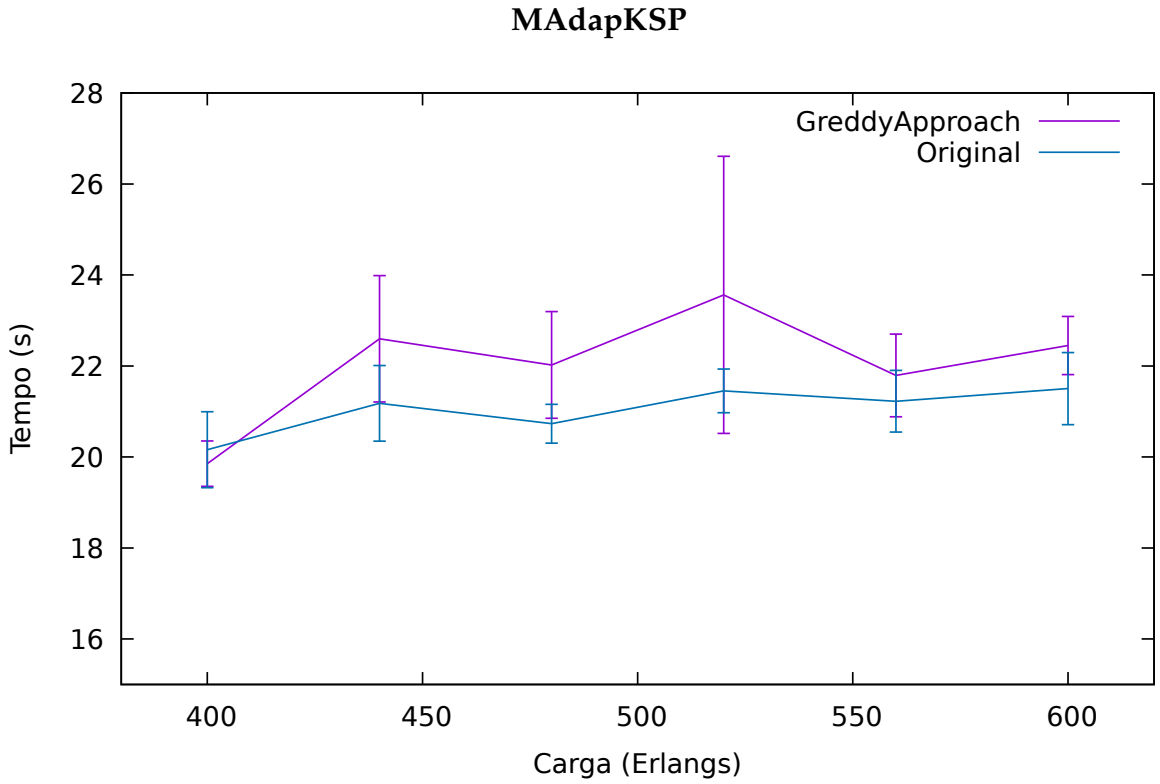

Figura 4.33: Tempo de execução do arquivo *EK.jar* no RA *MAdapKSP* executando no **cenário de 1 execução por carga**

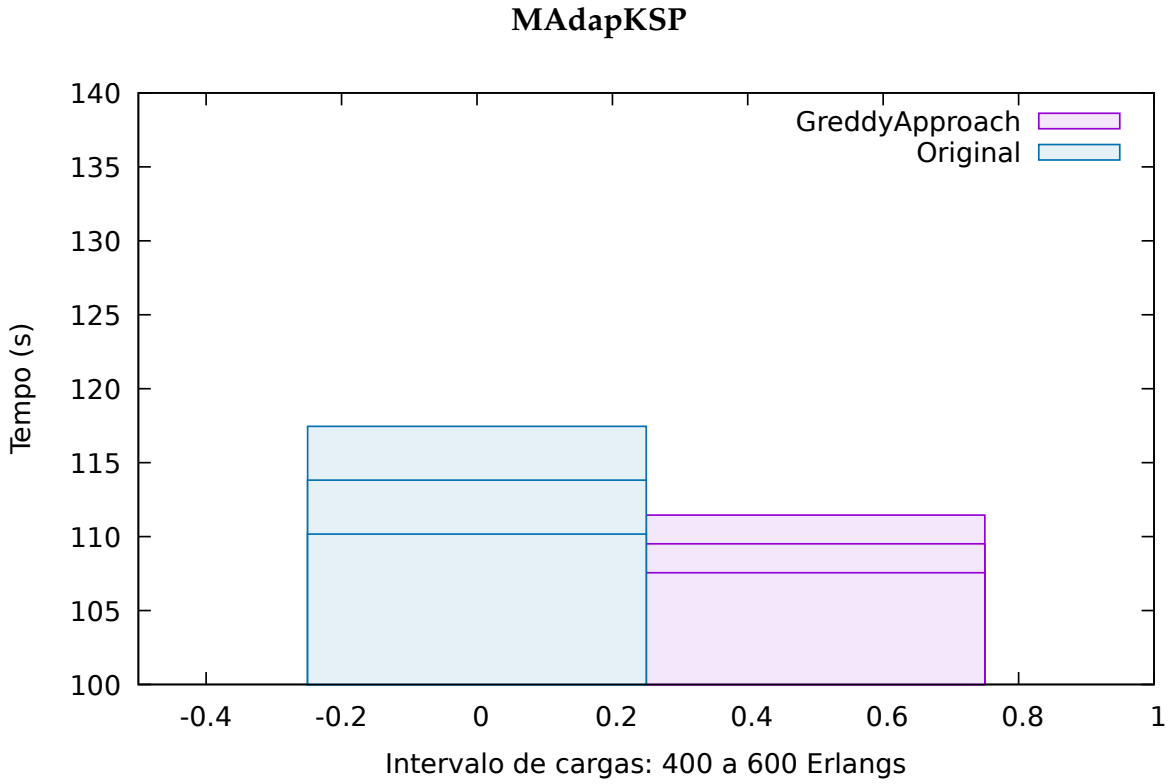

Figura 4.34: Tempo de execução do arquivo *EK.jar* no RA *MAdapKSP* executando no **cenário de várias cargas por execução**

Analisando a execução do *MAdapMSP* pode-se perceber uma leve melhoria na Figura 4.36 em relação a 4.32. Enquanto na carga única há cargas nas quais o Resizable-Array é melhor e outras nas quais a Greedy Approach é melhor.

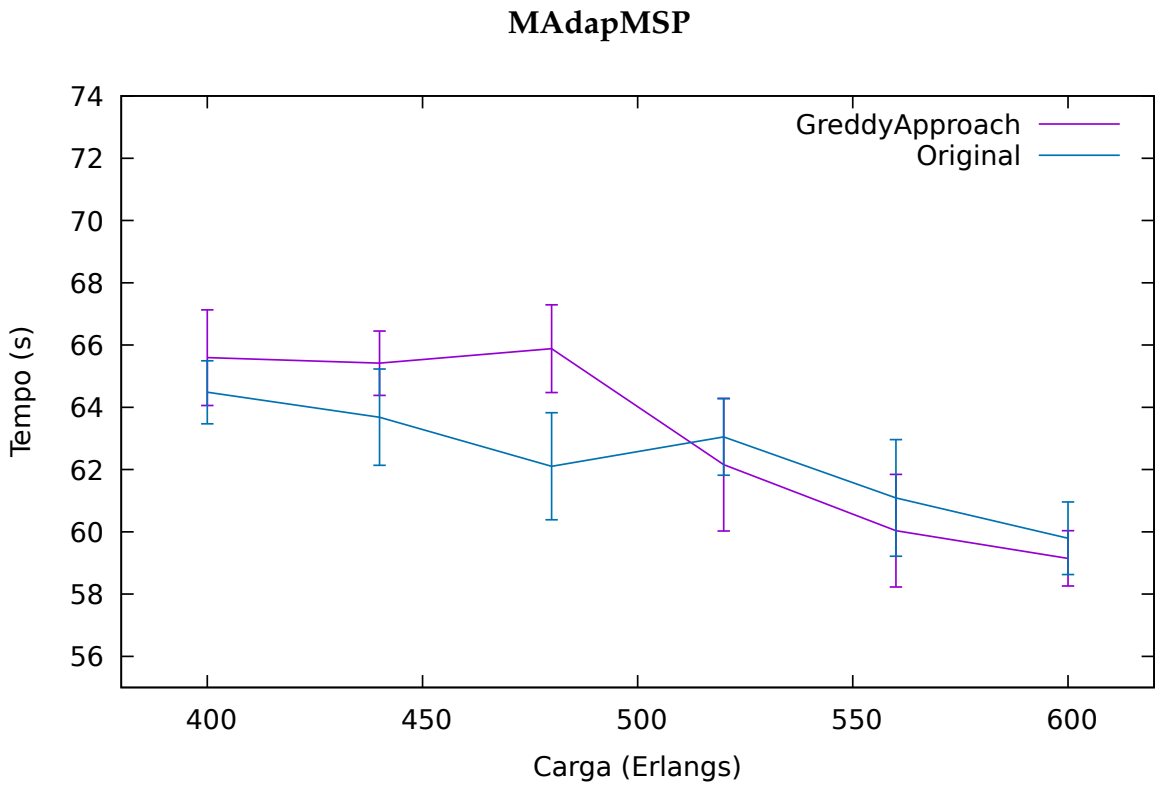

Figura 4.35: Tempo de execução do arquivo *EK.jar* no RA *MAdapMSP* executando no **cenário de 1 execução por carga**

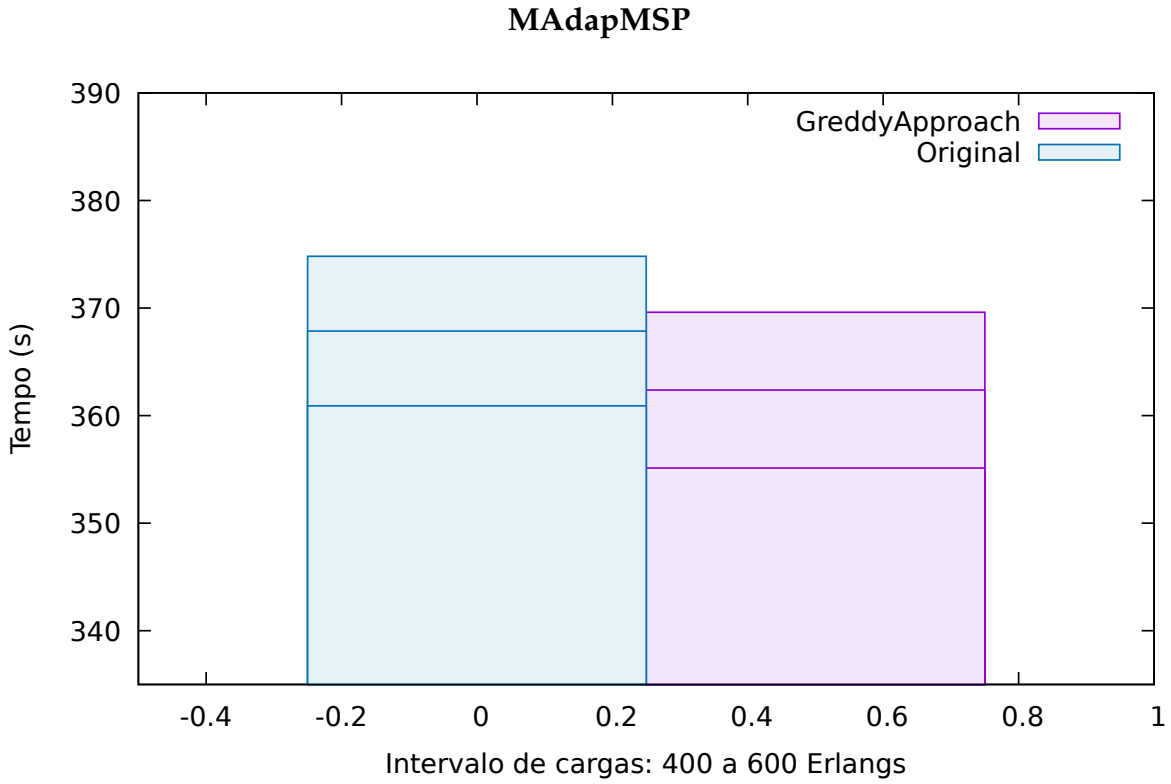

Figura 4.36: Tempo de execução do arquivo *EK.jar* no RA *MAdapMSP* executando no **cenário de várias cargas por execução**

Ao final dessa seção conclui-se que, enquanto essa melhoria apresentou bons resultados, esses resultados não são superiores a ideia do Resizable-Array, e ambos afetam a mesma função e, por isso não recomenda-se a sua implementação.

4.10 Resumo Conclusivo

Para concluir esse capítulo apresenta-se uma tabela resumo dos resultados obtidos, onde as propostas não recomendadas para ser incluídas no simulador estão em vermelho e as propostas recomendadas estão em verde.

As porcentagens foram calculados segundo a Equação 4.1.

$$perc = \frac{\bar{mod} - \bar{ori}}{\bar{ori}} * 100 \tag{4.1}$$

Onde $\bar{mod}$ é a média experimental obtida da modificação implementada e $\bar{ori}$ é a média experimental obtida da implementação original.

Tabela 4.1: Tabela resumo dos resultados da Seção **??**

	KSP 1 exec./carga	KSP várias cargas/exec.	MSP 1 exec./carga	MSP várias cargas/exec.
Lista de Adj.	De 3.87% até 13.9%	6.61%	De -2.69% até 4.89%	0.07%
Floyd-Warshall	De 2.00% até 10.74%	-1.93%	De 0.49% até 6.60%	-1.17%
Mult. de Matrizes	De -1.64% até 6.33%	-0.86%	De -1.35% até 3.20%	-3.56%
Min Heap	De 7.69% até 20.46%	6.08%	De -0.64% até 2.26%	-1.80%
Min Heap e Lista de Adj.	De 8.14% até 13.80%	9.54%	De -0.13% até 3.73%	-0.36%
Fibonacci Heap	De 0.87% até 9.85%	3.23%	De -1.08% até 4.62%	-0.76%
Fibonacci Heap e Lista de Adj.	De 6.08% até 11.92%	8.57%	De -0.68% até 3.83%	-2.89%
Resizable-Array	De -12.81% até -4.49%	-9.00%	De -0.96% até 3.91%	-0.63%
Greedy Approach	De -1.53% até 9.84%	-3.79%	De -1.41% até 6.08%	-1.50%

Assim uma **porcentagem negativa** indica que $\bar{mod} < \bar{ori}$, ou seja, a modificação leva **menos** tempo que a implementação original. Já uma **porcentagem positiva** indica que $\bar{mod} > \bar{ori}$, ou seja, a modificação leva **mais** tempo que a implementação original.

Adicionalmente, no cenário de 1 carga por execução, foram tabelados somente o menor e o maior resultado obtido, ou seja, para $\bar{mod}_1 < \bar{mod}_2 < \bar{mod}_3 < \bar{mod}_4 < \bar{mod}_5 < \bar{mod}_6$, foi escrito "De $\bar{mod}_1$ até $\bar{mod}_6$".

Capítulo 5

Conclusão

O simulador ONS, que permite a análise de tráfego dinâmico em redes com tecnologia de WDM e de EON, utiliza em sua implementação um conjunto de algoritmos computacionais que formam a base matemática de toda a simulação e cujo código afeta o tempo de execução da simulação como um todo, podendo gerar execuções bastante demoradas.

Em virtude disso, esse trabalho teve como objetivo a análise dos algoritmos implementados no ONS, propondo novas soluções ou melhorias aos algoritmos atuais de forma a diminuir o tempo de execução das simulações.

Para isso, fez-se uma análise das várias classes implementadas no simulador e verificou-se em quais casos é possível a troca de um algoritmo ou estrutura de dados por outra com melhor complexidade assintótica. Em seguida, fez-se um profiling do código para encontrar pontos de gargalo.

Obtendo esse conjunto de possíveis melhorias, elas foram implementadas e utilizou-se os algoritmos RSA, MAdapKSP e MAdapMSP para testá-las através do método das replicações independentes, após o qual, os resultados foram plotados

Na análise dos resultados, verificaram-se ganhos significativos nos algoritmos:

- Floyd-Warshall, onde há uma redução de até 1.93% no tempo e um aumento da simplicidade da implementação.
- Multiplicação de matrizes, onde há uma redução de até 3.56% no tempo e um aumento da simplicidade da implementação.
- Resizable-Array, onde há uma redução de até 12.81 % no tempo.

Logo, decidiu-se recomendá-los para adição ao simulador. Enquanto nos algoritmos a baixo, não é recomendada a adição ao simulador:

- Lista de Adjacências, onde há um aumento de até 13.9% no tempo e um aumento do intervalo de confiança, reduzindo a previsibilidade do tempo de execução.
- Min Heap, onde há um aumento de até 20.46% no tempo em alguns casos e quase nenhuma diferença em outros.
- Min Heap e Lista de Adjacências, onde há um aumento de até 13.80% no tempo.
- Fibonacci Heap, onde há um aumento de até 9.85% no tempo.
- Fibonacci Heap e Lista de Adjacências, onde há um aumento de até 11.92% no tempo em alguns casos, porém há uma redução de até 2.89% em outros.
- Greedy Approach, onde há uma redução de até 3.79%, porém essa modificação não pode ser implementada junta ao Resizable-Array (com até 12.81% de redução), uma vez que elas afetam a mesma função.

Com esse estudo, foi possível observar a diferença entre o estudo teórico e a análise prática. Em vários casos como no problema do menor caminho e na função *getSlotsAvailable*, escolheu-se ao final utilizar um algoritmo que possuía uma complexidade assintótica não ótima.

Isso pode ser explicado pelo fato de que, no contexto de redes ópticas elásticas, não se trabalha com um alto número de nós em um grafo ou *slots* de frequência em um nó, entre outros. Devido ao valor pequeno das variáveis que afetam a análise assintótica, a constante das soluções apresentadas acaba influenciando fortemente o tempo de execução.

Para concluir, dentre as mudanças propostas e implementadas no algoritmo decidiu-se implementar os algoritmos de Floyd-Warshall, o uso de Multiplicação de Matrizes nas funções de *clustering* e o uso de Resizable-Array na função *getSlotsAvaiable*.

Como trabalhos futuros, pode-se trabalhar na melhoria do problema do K shortest-paths através da utilização do algoritmo de Eppstein e pode-se portar o código para uma linguagem de programação com melhor desempenho como, por exemplo, C.

Referências

[1] Costa, L. R., Sousa L. S. de Oliveira F. R. de Silva K. A. da Júnior P. J. S. e Drummond A. C.: *Ons: Simulador de eventos discretos para redes Ópticas wdm/eon*. SBRC, 2016. x, 1, 29, 31, 32

[2] Jinno, Masahiko, Hidehiko Takara, Bartlomiej Kozicki, Yukio Tsukishima, Yoshiaki Sone e Shinji Matsuoka: *Spectrum-efficient and scalable elastic optical path network: Architecture, benefits, and enabling technologies*. Comm. Mag., 47(11):66–73, novembro 2009, ISSN 0163-6804. `http://dx.doi.org/10.1109/MCOM.2009.5307468`. 1

[3] Gerstel, Ori, Masahiko Jinno, Andrew Lord e S. J. Ben Yoo: *Elastic optical networking: a new dawn for the optical layer?* IEEE Communications Magazine, 50, 2012. 1

[4] Christodoulopoulos, K., Tomkos I. Varvarigos E.: *Elastic bandwidth allocation in flexible ofdm-based optical networks*. Journal of Lightwave Technology, 29(9):1354–1366, 2011. 1, 31

[5] Zhang, G., M. De Leenheer, A. Morea e B. Mukherjee: *A survey on ofdm-based elastic core optical networking*. IEEE Communications Surveys Tutorials, 15(1):65–87, First 2013, ISSN 1553-877X. 1

[6] Jinno, Masahiko, H Takara e B Kozicki: *Dynamic optical mesh networks: Drivers, challenges and solutions for the future, optical communication*. Volume 7, páginas 1 – 4, outubro 2009. 1

[7] Palmieri, F., U. Fiore e S. Ricciardi: *Simulnet: a wavelength-routed optical network simulation framework*. Em *2009 IEEE Symposium on Computers and Communications*, páginas 281–286, July 2009. 1, 40

[8] L.R. Costa, L.S. de Sousa, F.R. de Oliveira K.A. da Silva P.J.S. Júnior A.C. Drummond: *Ons: Optical network simulator—wdm/eon*. `https://gitlab.com/get-unb/ons`, 2016. 1

[9] Cormen, T. H. et al.: *Introduction to Algorithms*, volume 3rd ed. The MIT Press, MA, USA, 2009. 3, 6, 7, 9, 10

[10] Sipser, M.: *Introduction to the Theory of Computation*, volume 2nd ed. Thomson, USA, 2006. 3, 4

[11] Knuth, Donald E.: *Big omicron and big omega and big theta*. SIGACT News, 8(2):18–24, abril 1976, ISSN 0163-5700. `http://doi.acm.org/10.1145/1008328.1008329`. 5

[12] Goodrich, Michael T. e Roberto Tamassia: *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edição, 2009, ISBN 0470088540, 9780470088548. 9

[13] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. 10

[14] Windley, P. F.: *Trees, Forests and Rearranging*. The Computer Journal, 3(2):84–88, janeiro 1960, ISSN 0010-4620. https://doi.org/10.1093/comjnl/3.2.84. 10

[15] Williams, J. W. J.: *Algorithm 232: Heapsort*. Communications of the ACM, 7(6):347–348, 1964. 11

[16] Fredman, Michael L. e Robert Endre Tarjan: *Fibonacci heaps and their uses in improved network optimization algorithms*. J. ACM, 34(3):596–615, julho 1987, ISSN 0004-5411. http://doi.acm.org/10.1145/28869.28874. 14

[17] Dijkstra, E. W.: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1(1):269–271, Dec 1959, ISSN 0945-3245. https://doi.org/10.1007/BF01386390. 22

[18] Floyd, Robert W.: *Algorithm 97: Shortest path*. Commun. ACM, 5(6):345–, junho 1962, ISSN 0001-0782. http://doi.acm.org/10.1145/367766.368168. 25

[19] Warshall, Stephen: *A theorem on boolean matrices*. J. ACM, 9(1):11–12, janeiro 1962, ISSN 0004-5411. http://doi.acm.org/10.1145/321105.321107. 25

[20] Chatterjee, B. C., Sarma N. e E. Oki: *Routing and spectrum allocation in elastic optical networks: A tutorial*. IEEE Communication surveys & tutorials, 17(3):1776–1800, 2015. 29

[21] Costa, L. R. e A. C. Drummond: *New distance-adaptive modulation scheme for elastic optical networks*. IEEE Communications Letters, 21(2):282–285, Feb 2017, ISSN 1089-7798. 30, 31

[22] Costa, L. R., Ramos G. R. e Drummond A. C.: *Leveraging adaptive modulation with multi-hop routing in elastic optical networks*. Computer Networks, 105(4):124–137, 2016. 31, 33, 34, 46

[23] Soares, André, Gilvan Durães, William Giozza e Paulo Cunha: *Tonets: Simulador para avaliação de desempenho de redes Ópticas transparentes*. janeiro 2007. 31

[24] Palmieri, F., U. Fiore e S. Ricciardi: *Simulnet: a wavelength-routed optical network simulation framework*. Em *2009 IEEE Symposium on Computers and Communications*, páginas 281–286, July 2009. 31

[25] Wan, Xin, Lei Wang, Nan Hua, Hanyi Zhang e Xiaoping Zheng: *Dynamic routing and spectrum assignment in flexible optical path networks*. Em *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2011*, página JWA055. Optical Society of America, 2011. http://www.osapublishing.org/abstract.cfm?URI=NFOEC-2011-JWA055. 31

[26] Wan, Xin, Nan Hua e Xiaoping Zheng: *Dynamic routing and spectrum assignment in spectrum-flexible transparent optical networks*. J. Opt. Commun. Netw., 4(8):603–613, Aug 2012. http://jocn.osa.org/abstract.cfm?URI=jocn-4-8-603. 31

[27] Jain, Raj: *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley professional computing. Wiley, 1991, ISBN 978-0-471-50336-1. 39, 40

Apêndice A

Modificações de Código

A.1 Modificações na branch master

No algoritmo de Djikstra implementado originalmente, não tinha sido estabelecido um critério de desempate entre 2 caminhos que tivessem um mesmo peso. Enquanto isso não afeta a corretude do algoritmo, ela dificulta a avaliação da corretude das modificações feitas neste trabalho. Assim, a seguinte modificação foi feita na branch master:

```
diff --git a/src/ons/util/Dijkstra.java b/src/ons/util/Dijkstra.java
index e49c41ac8bf7..c5050b93ce99 100644
--- a/src/ons/util/Dijkstra.java
+++ b/src/ons/util/Dijkstra.java
@@ -55,7 +55,7 @@ public class Dijkstra {
                 for (int j = 0; j < n.length; j++) {
                     final int v = n[j];
                     final double d = dist[next] + G.getWeight(next,
                        v);
-                    if (dist[v] > d) {
+                    if (dist[v] > d || (dist[v] == d && next < pred[
    v])) {
                         dist[v] = d;
                         pred[v] = next;
                     }
```

A.2 Modificações na branch AdjList

```
diff --git a/src/ons/util/WeightedGraph.java b/src/ons/util/
   WeightedGraph.java
index f6e98b5b1696..1c2c22c58e08 100644
--- a/src/ons/util/WeightedGraph.java
+++ b/src/ons/util/WeightedGraph.java
@@ -7,6 +7,7 @@ package ons.util;
 import java.io.Serializable;
 import java.util.ArrayList;
 import java.util.TreeSet;
+import javafx.util.Pair;

 /**
  * A weighted graph associates a label (weight) with every edge in
     the graph.
@@ -18,7 +19,9 @@ import java.util.TreeSet;
 public class WeightedGraph implements Serializable{

     private int numNodes;
-    private double[][] edges;  // adjacency matrix
+    private ArrayList< Pair<Integer, Double> >[] edges;  //
   adjacency matrix
+
+    private ArrayList< Pair<Integer, Double> >[] opos_edges;  //
   adjacency matrix

     /**
      * Creates a new WeightedGraph object with no edges,
@@ -26,7 +29,12 @@ public class WeightedGraph implements Serializable
   {
      * @param n number of nodes the new graph will have
      */
     public WeightedGraph(int n) {
-        edges = new double[n][n];
+        edges = new ArrayList[n];
+        opos_edges = new ArrayList[n];
+        for (int i = 0; i < n; i++) {
+            edges[i] = new ArrayList<>();
+            opos_edges[i] = new ArrayList<>();
+        }
         numNodes = n;
```

```
     }

@@ -38,14 +46,28 @@ public class WeightedGraph implements
    Serializable{
       */
     public WeightedGraph(WeightedGraph g) {
         numNodes = g.numNodes;
-        edges = new double[numNodes][numNodes];
+        edges = new ArrayList[numNodes];
+        opos_edges = new ArrayList[numNodes];
         for (int i = 0; i < numNodes; i++) {
-            for (int j = 0; j < numNodes; j++) {
-                edges[i][j] = g.getWeight(i, j);
+            edges[i] = new ArrayList<>();
+            opos_edges[i] = new ArrayList<>();
+            for (int j = 0; j < g.numNeighbors(i); j++) {
+                edges[i].add(g.getEdgeIndex(i, j));
+            }
+            for (int j = 0; j < g.numOposNeighbors(i); j++) {
+                opos_edges[i].add(g.getOposEdgeIndex(i, j));
             }
         }
     }

+    public Pair<Integer, Double> getEdgeIndex(int i, int j) {
+        return edges[i].get(j);
+    }
+
+    public Pair<Integer, Double> getOposEdgeIndex(int i, int j) {
+        return opos_edges[i].get(j);
+    }
+
     /**
      * Retrieves the size of the graph, i.e., the amount of vertexes
          it has.
      *
@@ -55,6 +77,14 @@ public class WeightedGraph implements Serializable
    {
         return numNodes;
     }
```

```
+    public int numNeighbors(int i) {
+        return edges[i].size();
+    }
+
+    public int numOposNeighbors(int i) {
+        return opos_edges[i].size();
+    }
+
     /**
      * Creates a new edge within the graph, which requires its two
        vertexes
      * and its weight.
@@ -64,7 +94,15 @@ public class WeightedGraph implements Serializable
    {
      * @param w the value of the edge's weight
      */
     public void addEdge(int source, int target, double w) {
-        edges[source][target] = w;
+
+        if (w == 0.0)
+            return;
+
+        Pair<Integer, Double> edge = new Pair(target, w);
+        edges[source].add(edge);
+
+        edge = new Pair(source, w);
+        opos_edges[target].add(edge);
     }

     /**
@@ -75,7 +113,10 @@ public class WeightedGraph implements
   Serializable{
      * @return true if the edge exists, or false otherwise
      */
     public boolean isEdge(int source, int target) {
-        return edges[source][target] > 0;
+        for (int j = 0; j < edges[source].size(); j++)
+            if (edges[source].get(j).getKey() == target)
+                return true;
```

```
+         return false;
      }

      /**
@@ -86,7 +127,12 @@ public class WeightedGraph implements
    Serializable{
       * @param target the edge's destination node
       */
      public void removeEdge(int source, int target) {
-         edges[source][target] = 0;
+         for (int j = 0; j < edges[source].size(); j++)
+             if (edges[source].get(j).getKey() == target)
+                 edges[source].remove(j);
+         for (int j = 0; j < opos_edges[target].size(); j++)
+             if (opos_edges[target].get(j).getKey() == source)
+                 opos_edges[target].remove(j);
      }

      /**
@@ -97,7 +143,11 @@ public class WeightedGraph implements
    Serializable{
       * @return the value of the edge's weight
       */
      public double getWeight(int source, int target) {
-         return edges[source][target];
+         for (int j = 0; j < edges[source].size(); j++)
+             if (edges[source].get(j).getKey() == target)
+                 return edges[source].get(j).getValue();
+
+         return 0.0;
      }

      /**
@@ -108,7 +158,29 @@ public class WeightedGraph implements
    Serializable{
       * @param w the value of the weight
       */
      public void setWeight(int source, int target, double w) {
-         edges[source][target] = w;
+         Pair<Integer, Double> p = new Pair(target, w);
```

```
+
+        if (w == 0.0) {
+            removeEdge(source, target);
+            return;
+        }
+
+        int j;
+        for (j = 0; j < edges[source].size(); j++)
+            if (edges[source].get(j).getKey() == target)
+                edges[source].set(j, p);
+        if (j == edges[source].size()) {
+            edges[source].add(p);
+        }
+
+        p = new Pair(source, w);
+
+        for (j = 0; j < opos_edges[target].size(); j++)
+            if (opos_edges[target].get(j).getKey() == source)
+                opos_edges[target].set(j, p);
+        if (j == opos_edges[target].size()) {
+            opos_edges[target].add(p);
+        }
     }

     /**
@@ -118,20 +190,10 @@ public class WeightedGraph implements
   Serializable{
       * @return list with indexes of the vertex's neighbors
       */
     public int[] neighbors(int vertex) {
-        int count = 0;
-        for (int i = 0; i < edges[vertex].length; i++) {
-            if (edges[vertex][i] > 0) {
-                count++;
-            }
-        }
-        final int[] answer = new int[count];
-        count = 0;
-        for (int i = 0; i < edges[vertex].length; i++) {
-            if (edges[vertex][i] > 0) {
```

```
-                 answer[count++] = i;
-             }
-         }
-         return answer;
+         int[] resp = new int[edges[vertex].size()];
+         for (int j = 0; j < edges[vertex].size(); j++)
+             resp[j] = edges[vertex].get(j).getKey();
+         return resp;
      }

      /**
@@ -141,20 +203,20 @@ public class WeightedGraph implements
    Serializable{
       * @return list with indexes of the vertex's neighbors
       */
      public int[] neighbors2(int vertex) {
-         int count = 0;
-         for (int i = 0; i < edges[vertex].length; i++) {
-             if (edges[i][vertex] > 0) {
-                 count++;
-             }
-         }
-         final int[] answer = new int[count];
-         count = 0;
-         for (int i = 0; i < edges[vertex].length; i++) {
-             if (edges[i][vertex] > 0) {
-                 answer[count++] = i;
-             }
-         }
-         return answer;
+         int[] resp = new int[opos_edges[vertex].size()];
+         for (int j = 0; j < opos_edges[vertex].size(); j++)
+             resp[j] = opos_edges[vertex].get(j).getKey();
+         return resp;
+     }
+
+     /**
+      * Retrieves the neighbors of a given vertex, and the weight
    between them.
+      * The vertices that are reachable by this vertex
```

```
+     * @param vertex index of the vertex within the matrix of edges
+     * @return list with indexes of the vertex's neighbors
+     */
+    public ArrayList< Pair<Integer, Double> > neighborsComplete(int
   vertex) {
+        return edges[vertex];
     }

     /**
@@ -169,10 +231,8 @@ public class WeightedGraph implements
   Serializable{
         String s = "";
         for (int j = 0; j < edges.length; j++) {
             s += Integer.toString(j) + ": ";
-            for (int i = 0; i < edges[j].length; i++) {
-                if (edges[j][i] > 0) {
-                    s += Integer.toString(i) + ":" + Double.toString
   (edges[j][i]) + " ";
-                }
+            for (int i = 0; i < edges[j].size(); i++) {
+                s += Integer.toString(i) + ":" + Double.toString(
   edges[j].get(i).getKey()) + " ";
             }
             s += "\n";
         }
@@ -205,34 +265,36 @@ public class WeightedGraph implements
   Serializable{
         //removing edges from this node
         removeNodeEdge(node);
         //remove node from the graph
-        double[][] newedges = new double[numNodes-1][numNodes-1];
+
+        ArrayList< Pair<Integer, Double> >[] newedges = new
   ArrayList[numNodes-1];
+        ArrayList< Pair<Integer, Double> >[] newopos_edges = new
  ArrayList[numNodes-1];
         int k = 0, l = 0;
-        for(int i = 0; i< numNodes; i++){
+        for(int i = 0; i < numNodes; i++){
             if(i != node){
```

```
-                 l = 0;
-                 for(int j = 0; j< numNodes; j++){
-                     if(j != node){
-                         newedges[k][l] = edges[i][j];
-                         l++;
-                     }
-                 }
-                 k++;
+                 newedges[k] = edges[i];
+                 newopos_edges[k] = opos_edges[i];
+                 k++;
             }
         }
         numNodes--;
         edges = newedges;
+        opos_edges = newopos_edges;
     }

     /**
      * Creates a new node in graph.
      */
     public void addNode(){
-        double[][] newedges = new double[numNodes+1][numNodes+1];
+
+        ArrayList< Pair<Integer, Double> >[] newedges = new
   ArrayList[numNodes+1];
+        ArrayList< Pair<Integer, Double> >[] newopos_edges = new
   ArrayList[numNodes+1];
         for(int i = 0; i < numNodes; i++){
-            System.arraycopy(edges[i], 0, newedges[i], 0, numNodes);
+            newedges[i] = edges[i];
+            newopos_edges[i] = opos_edges[i];
         }
         numNodes++;
         edges = newedges;
+        opos_edges = newopos_edges;
     }

     /**
```

A.3 Modificações na branch FloydWarshall

```
diff --git a/src/ons/util/WeightedGraph.java b/src/ons/util/
   WeightedGraph.java
index f6e98b5b1696..90b77ce1bd46 100644
--- a/src/ons/util/WeightedGraph.java
+++ b/src/ons/util/WeightedGraph.java
@@ -20,6 +20,8 @@ public class WeightedGraph implements Serializable{
     private int numNodes;
     private double[][] edges;  // adjacency matrix

+    static double oo = 1e18;
+
     /**
      * Creates a new WeightedGraph object with no edges,
      *
@@ -240,22 +242,36 @@ public class WeightedGraph implements
   Serializable{
      * @return the longest of all the calculated shortest paths in a
          network
      */
     public double getGraphDiameter(){
-        double max = 0, min = 0;
-        int[] nodes;
+
+        double[][] dp = new double[numNodes][numNodes];
         for (int i = 0; i < numNodes; i++) {
             for (int j = 0; j < numNodes; j++) {
-                if (i != j) {
-                    nodes = Dijkstra.getShortestPath(this, i, j);
-                    for (int k = 0; k < nodes.length - 1; k++) {
-                        min += getWeight(nodes[k], nodes[k + 1]);
+                    if (isEdge(i, j)) {
+                        dp[i][j] = getWeight(i, j);
+                    } else {
+                        dp[i][j] = oo;
                     }
-                    if(min > max){
-                        max = min;
-                    }
-                }
+            }
+        }
+        for (int k = 0; k < numNodes; k++) {
+            for (int i = 0; i < numNodes; i++) {
+                for (int j = 0; j < numNodes; j++) {
+                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j]);
+                }
+            }
+        }
+        double max = 0;
+        for (int i = 0; i < numNodes; i++) {
+            for (int j = 0; j < numNodes; j++) {
+                max = Math.max(max, dp[i][j]);
             }
         }
         return max;
```

```
+         }
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 for (int k = 0; k < numNodes; k++) {
+                     if (dp[i][k] + dp[k][j] < dp[i][j]) {
+                         dp[i][j] = dp[i][k] + dp[k][j];
                      }
                  }
-                 min = 0;
              }
          }
+         double max = 0;
+
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                     if (dp[i][j] < oo && max < dp[i][j]) {
+                         max = dp[i][j];
+                     }
+             }
+         }
+
          return max;
      }

@@ -366,6 +382,41 @@ public class WeightedGraph implements
    Serializable{
       * @return the average path length of this graph
       */
      public double getAveragePathLength(){
+
+         double[][] dp = new double[numNodes][numNodes];
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                     if (isEdge(i, j)) {
+                         dp[i][j] = getWeight(i, j);
+                     } else {
+                         dp[i][j] = oo;
+                     }
+             }
+         }
```

```
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 for (int k = 0; k < numNodes; k++) {
+                     if (dp[i][k] + dp[k][j] < dp[i][j]) {
+                         dp[i][j] = dp[i][k] + dp[k][j];
+                     }
+                 }
+             }
+         }
+         double sumDistance = 0;
+
+         System.out.println("Executado\n");
+
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                     if (i != j) {
+                         if (dp[i][j] < oo)
+                             sumDistance += dp[i][j];
+                     }
+             }
+         }
+         return sumDistance/ (double) (numNodes * (numNodes - 1));
+
+         /*
          double sumDistance = 0;
          int[] nodes;
          int nodePairs = 0;
@@ -381,6 +432,7 @@ public class WeightedGraph implements
    Serializable{
                  }
          }
          return sumDistance/ (double) nodePairs;
+         */
      }

      /**
```

A.4 Modificações na branch Clustering

```
diff --git a/src/ons/util/WeightedGraph.java b/src/ons/util/
   WeightedGraph.java
index f6e98b5b1696..b4f84653b55f 100644
--- a/src/ons/util/WeightedGraph.java
+++ b/src/ons/util/WeightedGraph.java
@@ -327,37 +327,51 @@ public class WeightedGraph implements
   Serializable{
      * @return the global clustering coefficient
      */
     public double getGlobalClusteringCoefficient(){
-        ArrayList<TreeSet<Integer>> triangles = new ArrayList<>();
-        ArrayList<TreeSet<Integer>> triplets = new ArrayList<>();
-        TreeSet<Integer> triangle;
-        TreeSet<Integer> triplet;
-        for (int node = 0; node < numNodes; node++) {
-            int[] neighbors = neighbors(node);
-            for (int i = 0; i < neighbors.length; i++) {
-                for (int j = 0; j < neighbors.length; j++) {
-                    if (neighbors[i] != neighbors[j]) {
-                        triplet = new TreeSet<>();
-                        triplet.add(node);
-                        triplet.add(neighbors[i]);
-                        triplet.add(neighbors[j]);
-                        if (!triplets.contains(triplet)) {
-                            triplets.add(triplet);
-                        }
-                        if (isEdge(neighbors[i], neighbors[j])) {
-                            triangle = new TreeSet<>();
-                            triangle.add(node);
-                            triangle.add(neighbors[i]);
-                            triangle.add(neighbors[j]);
-                            if(!triangles.contains(triangle)){
-                                triangles.add(triangle);
-                            }
-                        }
-                    }
+
+        int[][] A = new int[numNodes][numNodes];
+        int[][] B = new int[numNodes][numNodes];
+
```

```
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 if (isEdge(i, j)) {
+                     A[i][j] = 1;
+                 } else {
+                     A[i][j] = 0;
+                 }
+                 B[i][j] = 0;
+             }
+         }
+
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 for (int k = 0; k < numNodes; k++) {
+                     B[i][j] += A[i][k] * A[k][j];
+                 }
+             }
+         }
+
+         int triangles = 0;
+         int triplets = 0;
+
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 if (A[i][j] == 1) {
+                     triangles += B[i][j];
+                 }
+             }
+         }
+         triangles /= 6;
+
+         for (int i = 0; i < numNodes; i++) {
+             for (int j = 0; j < numNodes; j++) {
+                 if (i != j) {
+                     triplets += B[i][j];
                  }
-
              }
          }
-         return (double) triangles.size() / (double) triplets.size();
```

```
+         triplets /= 2;
+         triplets -= 2 * triangles;
+         return (double) triangles / (double) triplets;
      }

      /**
```

A.5 Modificações na branch MinHeap

```
diff --git a/src/ons/util/Dijkstra.java b/src/ons/util/Dijkstra.java
index c5050b93ce99..16cbedfb7c62 100644
--- a/src/ons/util/Dijkstra.java
+++ b/src/ons/util/Dijkstra.java
@@ -33,21 +33,31 @@ public class Dijkstra {
      public static int[] dijkstra(WeightedGraph G, int s) {
          final double[] dist = new double[G.size()];  // shortest
             known distance from "s"
          final int[] pred = new int[G.size()];  // preceding node in
             path
-         final boolean[] visited = new boolean[G.size()]; // all
    false initially
+         final MinHeap visited = new MinHeap();

          for (int i = 0; i < dist.length; i++) {
              pred[i] = -1;
              dist[i] = Integer.MAX_VALUE;
          }
          dist[s] = 0;
-
-         for (int i = 0; i < dist.length; i++) {
-             final int next = minVertex(dist, visited);
+
+         visited.pushBack(dist[s], s);
+
+         while(visited.size() > 0) {
+             final int next = visited.topKey();
+             final double nextDist = visited.topHeap();
+
+             visited.pop();
+
```

```
 //             if (next < 0) {
 //                 return pred;
 //             }
             if (next >= 0) {
-                visited[next] = true;
+
+                if (nextDist > dist[next]) { //Elemento do heap
   desatualizado
+                    continue;
+                }
+


                 // The shortest path to next is dist[next] and via
                    pred[next].

@@ -58,6 +68,8 @@ public class Dijkstra {
                     if (dist[v] > d || (dist[v] == d && next < pred[
                        v])) {
                         dist[v] = d;
                         pred[v] = next;
+
+                        visited.pushBack(dist[v], v);
                     }
                 }
             }
@@ -65,27 +77,6 @@ public class Dijkstra {
         return pred;  // (ignore pred[s]==0!)
     }

-    /**
-     * Finds, from the list of unvisited vertexes, the one with the
   lowest
-     * distance from the initial node.
-     *
-     * @param dist vector with shortest known distance from the
   initial node
-     * @param v vector indicating the visited nodes
-     * @return vertex with minimum distance from initial node,
-     *                          or -1 if the graph is unconnected or
   if no vertexes were visited yet
```

```
-     */
-    private static int minVertex(double[] dist, boolean[] v) {
-        double x = Double.MAX_VALUE;
-        int y = -1;   // graph not connected, or no unvisited
    vertices
-        for (int i = 0; i < dist.length; i++) {
-            if (!v[i] && dist[i] < x) {
-                y = i;
-                x = dist[i];
-            }
-        }
-        return y;
-    }
-
     /**
      * Retrieves the shortest path between a source and a
        destination node,
     * within a weighted graph.
@@ -173,21 +164,30 @@ public class Dijkstra {
     public static int[] dijkstra(WeightedMultiGraph G, int s) {
         final double[] dist = new double[G.size()];  // shortest
           known distance from "s"
         final int[] pred = new int[G.size()];  // preceding node in
           path
-        final boolean[] visited = new boolean[G.size()]; // all
    false initially
+        MinHeap visited = new MinHeap();

         for (int i = 0; i < dist.length; i++) {
             pred[i] = -1;
             dist[i] = Integer.MAX_VALUE;
         }
         dist[s] = 0;
+
+        visited.pushBack(dist[s], s);

-        for (int i = 0; i < dist.length; i++) {
-            final int next = minVertex(dist, visited);
+        while(visited.size() > 0) {
+            final int next = visited.topKey();
```

```
+             final double nextDist = visited.topHeap();
 //             if (next < 0) {
 //                 return pred;
 //             }
+
+             visited.pop();
+
              if (next >= 0) {
-                 visited[next] = true;
+
+                 if (nextDist > dist[next]) {
+                     continue;
+                 }

                  // The shortest path to next is dist[next] and via
                     pred[next].

@@ -198,6 +198,8 @@ public class Dijkstra {
                      if (dist[v] > d) {
                          dist[v] = d;
                          pred[v] = next;
+
+                         visited.pushBack(dist[v], v);
                      }
                  }
              }
@@ -243,7 +245,7 @@ public class Dijkstra {
      public static int[] dijkstra(WeightedMultiGraphMultiWeight G,
         int s, int index) {
          final double[] dist = new double[G.size()];  // shortest
             known distance from "s"
          final int[] pred = new int[G.size()];  // preceding node in
             path
-         final boolean[] visited = new boolean[G.size()]; // all
    false initially
+         final MinHeap visited = new MinHeap(); // all false
    initially

          for (int i = 0; i < dist.length; i++) {
              pred[i] = -1;
```

```
@@ -251,13 +253,22 @@ public class Dijkstra {
         }
         dist[s] = 0;

-        for (int i = 0; i < dist.length; i++) {
-            final int next = minVertex(dist, visited);
+        visited.pushBack(dist[s], s);
+
+        while(visited.size() > 0) {
+            final int next = visited.topKey();
+            final double nextDist = visited.topHeap();
 //            if (next < 0) {
 //                return pred;
 //            }
+
+            visited.pop();
+
             if (next >= 0) {
-                visited[next] = true;
+
+                if (nextDist > dist[next]) {
+                    continue;
+                }

                 // The shortest path to next is dist[next] and via
                    pred[next].

@@ -268,6 +279,8 @@ public class Dijkstra {
                     if (dist[v] > d) {
                         dist[v] = d;
                         pred[v] = next;
+
+                        visited.pushBack(dist[v], v);
                     }
                 }
             }
@@ -314,21 +327,30 @@ public class Dijkstra {
     public static int[] dijkstra(WeightedGraphMultiWeight G, int s,
        int index) {
```

```
          final double[] dist = new double[G.size()];  // shortest
             known distance from "s"
          final int[] pred = new int[G.size()];  // preceding node in
             path
-         final boolean[] visited = new boolean[G.size()]; // all
    false initially
+         final MinHeap visited = new MinHeap(); // all false
    initially

          for (int i = 0; i < dist.length; i++) {
              pred[i] = -1;
              dist[i] = Integer.MAX_VALUE;
          }
          dist[s] = 0;
+
+         visited.pushBack(dist[s], s);

-         for (int i = 0; i < dist.length; i++) {
-             final int next = minVertex(dist, visited);
+         while(visited.size() > 0) {
+             final int next = visited.topKey();
+             final double nextDist = visited.topHeap();
 //              if (next < 0) {
 //                  return pred;
 //              }
+
+             visited.pop();
+
              if (next >= 0) {
-                 visited[next] = true;
+
+                 if (nextDist > dist[next]) {
+                     continue;
+                 }

                  // The shortest path to next is dist[next] and via
                     pred[next].

@@ -339,6 +361,8 @@ public class Dijkstra {
                      if (dist[v] > d) {
```

```
                        dist[v] = d;
                        pred[v] = next;
+
+                       visited.pushBack(dist[v], v);
                    }
                }
            }
diff --git a/src/ons/util/MinHeap.java b/src/ons/util/MinHeap.java
new file mode 100644
index 000000000000..682910ddf591
--- /dev/null
+++ b/src/ons/util/MinHeap.java
@@ -0,0 +1,146 @@
+/*
+ * To change this license header, choose License Headers in Project
   Properties.
+ * To change this template file, choose Tools | Templates
+ * and open the template in the editor.
+ */
+package ons.util;
+
+import java.util.ArrayList;
+
+/**
+ *
+ * @author rchehab
+ */
+public class MinHeap {
+    ArrayList<Double> heap;
+    ArrayList<Integer> key;
+    int numElem;
+
+    public MinHeap() {
+        numElem = 0;
+        heap = new ArrayList<>();
+        key = new ArrayList<>();
+    }
+
+    public MinHeap(int numElem, int[] key, double[] heap) {
+        this.numElem = 0;
```

```
+        this.heap = new ArrayList<>();
+        this.key = new ArrayList<>();
+
+        for (int i = 0; i < numElem; i++) {
+            pushBack(heap[i], key[i]);
+        }
+    }
+
+    private int leftSon(int i) {
+        return 2 * i + 1;
+    }
+
+    private int rightSon(int i) {
+        return 2 * i + 2;
+    }
+
+    private int parent (int i) {
+        return (i - 1) / 2;
+    }
+
+    private void update(int i) {
+
+        if (i != 0) { //Verifica se o pai desse vertice e maior que
   ele
+
+            int par = parent(i);
+
+            if (heap.get(i) < heap.get(par)) {
+                double tempHeap = heap.get(i);
+                int tempKey = key.get(i);
+                heap.set(i, heap.get(par));
+                key.set(i, key.get(par));
+                heap.set(par, tempHeap);
+                key.set(par, tempKey);
+
+                update(par);
+            }
+        }
+        int left = leftSon(i);
+        int right = rightSon(i);
```

```
+        if (left < numElem) {
+            if (right < numElem) { //Possui ambos os filhos
+                int indexMin;
+
+                if (heap.get(left) < heap.get(right)) {
+                    indexMin = left;
+                } else {
+                    indexMin = right;
+                }
+
+                if (heap.get(indexMin) < heap.get(i)) {
+                    double tempHeap = heap.get(i);
+                    int tempKey = key.get(i);
+                    heap.set(i, heap.get(indexMin));
+                    key.set(i, key.get(indexMin));
+                    heap.set(indexMin, tempHeap);
+                    key.set(indexMin, tempKey);
+
+                    update(indexMin);
+                }
+            } else { //Possui somente o fiho da esquerda
+                if (heap.get(left) < heap.get(i)) {
+                    double tempHeap = heap.get(i);
+                    int tempKey = key.get(i);
+                    heap.set(i, heap.get(left));
+                    key.set(i, key.get(left));
+                    heap.set(left, tempHeap);
+                    key.set(left, tempKey);
+
+                    update(left);
+                }
+            }
+        }
+    }
+
+    public void pushBack(double heapElem, int keyElem) {
+
+        numElem++;
+
+        heap.add(numElem - 1, heapElem);
```

```
+        key.add(numElem - 1, keyElem);
+
+        update(numElem - 1);
+    }
+
+    public int size() {
+        return numElem;
+    }
+
+    public double topHeap() {
+        if (numElem > 0) {
+            return heap.get(0);
+        } else {
+            return -1;
+        }
+    }
+    public int topKey() {
+        if (numElem > 0) {
+            return key.get(0);
+        } else {
+            return -1;
+        }
+    }
+    public void pop() {
+
+        if (numElem == 0) {
+            return;
+        }
+
+        numElem--;
+
+        if (numElem == 0) {
+            return;
+        }
+        //Coloca o ultimo elemento no 0
+        heap.set(0, heap.get(numElem));
+        key.set(0, key.get(numElem));
+
+        update(0);
+    }
```

```
+}
```

A.6 Modificações na branch FibonacciHeap

```
diff --git a/src/ons/util/Dijkstra.java b/src/ons/util/Dijkstra.java
index c5050b93ce99..9371e767e004 100644
--- a/src/ons/util/Dijkstra.java
+++ b/src/ons/util/Dijkstra.java
@@ -33,21 +33,32 @@ public class Dijkstra {
     public static int[] dijkstra(WeightedGraph G, int s) {
         final double[] dist = new double[G.size()];  // shortest
            known distance from "s"
         final int[] pred = new int[G.size()];  // preceding node in
            path
-        final boolean[] visited = new boolean[G.size()]; // all
   false initially
+        final FibonacciHeap visited = new FibonacciHeap();

         for (int i = 0; i < dist.length; i++) {
             pred[i] = -1;
             dist[i] = Integer.MAX_VALUE;
         }
         dist[s] = 0;
-
-        for (int i = 0; i < dist.length; i++) {
-            final int next = minVertex(dist, visited);
+
+        visited.insert(dist[s], s);
+
+        while(visited.size() > 0) {
+
+            final int next = visited.findMinIndex();
+            final double nextDist = visited.findMinKey();
+
+            visited.deleteMin();
+
 //            if (next < 0) {
 //                return pred;
 //            }
             if (next >= 0) {
```

```
-                 visited[next] = true;
+
+                 if (nextDist > dist[next]) { //Elemento do heap
    desatualizado
+                     continue;
+                 }
+

                  // The shortest path to next is dist[next] and via
                     pred[next].

@@ -58,6 +69,8 @@ public class Dijkstra {
                      if (dist[v] > d || (dist[v] == d && next < pred[
                         v])) {
                          dist[v] = d;
                          pred[v] = next;
+
+                         visited.insert(dist[v], v);
                      }
                  }
              }
@@ -65,27 +78,6 @@ public class Dijkstra {
          return pred;  // (ignore pred[s]==0!)
      }

-     /**
-      * Finds, from the list of unvisited vertexes, the one with the
    lowest
-      * distance from the initial node.
-      *
-      * @param dist vector with shortest known distance from the
    initial node
-      * @param v vector indicating the visited nodes
-      * @return vertex with minimum distance from initial node,
-      *                           or -1 if the graph is unconnected or
    if no vertexes were visited yet
-      */
-     private static int minVertex(double[] dist, boolean[] v) {
-         double x = Double.MAX_VALUE;
```

```
-        int y = -1;   // graph not connected, or no unvisited
   vertices
-        for (int i = 0; i < dist.length; i++) {
-            if (!v[i] && dist[i] < x) {
-                y = i;
-                x = dist[i];
-            }
-        }
-        return y;
-    }
-
     /**
      * Retrieves the shortest path between a source and a
        destination node,
     * within a weighted graph.
@@ -173,21 +165,30 @@ public class Dijkstra {
     public static int[] dijkstra(WeightedMultiGraph G, int s) {
         final double[] dist = new double[G.size()];  // shortest
           known distance from "s"
         final int[] pred = new int[G.size()];  // preceding node in
           path
-        final boolean[] visited = new boolean[G.size()]; // all
   false initially
+        FibonacciHeap visited = new FibonacciHeap();

         for (int i = 0; i < dist.length; i++) {
             pred[i] = -1;
             dist[i] = Integer.MAX_VALUE;
         }
         dist[s] = 0;
+
+        visited.insert(dist[s], s);

-        for (int i = 0; i < dist.length; i++) {
-            final int next = minVertex(dist, visited);
+        while(visited.size() > 0) {
+            final int next = visited.findMinIndex();
+            final double nextDist = visited.findMinKey();
 //            if (next < 0) {
 //                return pred;
```

```
 //             }
+
+             visited.deleteMin();
+
              if (next >= 0) {
-                 visited[next] = true;
+
+                 if (nextDist > dist[next]) {
+                     continue;
+                 }

                  // The shortest path to next is dist[next] and via
                     pred[next].

@@ -198,6 +199,8 @@ public class Dijkstra {
                      if (dist[v] > d) {
                          dist[v] = d;
                          pred[v] = next;
+
+                         visited.insert(dist[v], v);
                      }
                  }
              }
@@ -243,7 +246,7 @@ public class Dijkstra {
      public static int[] dijkstra(WeightedMultiGraphMultiWeight G,
         int s, int index) {
          final double[] dist = new double[G.size()];  // shortest
             known distance from "s"
          final int[] pred = new int[G.size()];  // preceding node in
             path
-         final boolean[] visited = new boolean[G.size()]; // all
    false initially
+         final FibonacciHeap visited = new FibonacciHeap(); // all
    false initially

          for (int i = 0; i < dist.length; i++) {
              pred[i] = -1;
@@ -251,13 +254,22 @@ public class Dijkstra {
          }
          dist[s] = 0;
```

```
-         for (int i = 0; i < dist.length; i++) {
-             final int next = minVertex(dist, visited);
+         visited.insert(dist[s], s);
+
+         while(visited.size() > 0) {
+             final int next = visited.findMinIndex();
+             final double nextDist = visited.findMinKey();
 //              if (next < 0) {
 //                  return pred;
 //              }
+
+             visited.deleteMin();
+
              if (next >= 0) {
-                 visited[next] = true;
+
+                 if (nextDist > dist[next]) {
+                     continue;
+                 }

                  // The shortest path to next is dist[next] and via
                     pred[next].

@@ -268,6 +280,8 @@ public class Dijkstra {
                      if (dist[v] > d) {
                          dist[v] = d;
                          pred[v] = next;
+
+                         visited.insert(dist[v], v);
                      }
                  }
              }
@@ -314,21 +328,30 @@ public class Dijkstra {
      public static int[] dijkstra(WeightedGraphMultiWeight G, int s,
         int index) {
          final double[] dist = new double[G.size()];  // shortest
             known distance from "s"
          final int[] pred = new int[G.size()];  // preceding node in
             path
```

```
-        final boolean[] visited = new boolean[G.size()]; // all
    false initially
+        final FibonacciHeap visited = new FibonacciHeap(); // all
    false initially

         for (int i = 0; i < dist.length; i++) {
             pred[i] = -1;
             dist[i] = Integer.MAX_VALUE;
         }
         dist[s] = 0;
+
+        visited.insert(dist[s], s);

-        for (int i = 0; i < dist.length; i++) {
-            final int next = minVertex(dist, visited);
+        while(visited.size() > 0) {
+            final int next = visited.findMinIndex();
+            final double nextDist = visited.findMinKey();
 //            if (next < 0) {
 //                return pred;
 //            }
+
+            visited.deleteMin();
+
             if (next >= 0) {
-                visited[next] = true;
+
+                if (nextDist > dist[next]) {
+                    continue;
+                }

                 // The shortest path to next is dist[next] and via
                    pred[next].

@@ -339,6 +362,8 @@ public class Dijkstra {
                     if (dist[v] > d) {
                         dist[v] = d;
                         pred[v] = next;
+
+                        visited.insert(dist[v], v);
```

```
                    }
                }
            }
diff --git a/src/ons/util/FibonacciHeap.java b/src/ons/util/
   FibonacciHeap.java
new file mode 100644
index 000000000000..746901c931ca
--- /dev/null
+++ b/src/ons/util/FibonacciHeap.java
@@ -0,0 +1,213 @@
+/*
+ * To change this license header, choose License Headers in Project
   Properties.
+ * To change this template file, choose Tools | Templates
+ * and open the template in the editor.
+ */
+package ons.util;
+
+/**
+ *
+ * @author rchehab
+ */
+public class FibonacciHeap {
+
+    FibonacciNode min;
+
+    int n;
+
+    public FibonacciHeap() {
+        min = null;
+        n = 0;
+    }
+
+    public void insert(double key, int index) {
+        FibonacciNode newKey = new FibonacciNode(key, index);
+
+        if (newKey == null) {
+            return;
+        }
+
```

```
+        n++;
+
+        if (min == null) {
+            min = newKey;
+            return;
+        }
+
+        min.insertRight(newKey);
+
+        if (newKey.getKey() < min.getKey()) {
+            min = newKey;
+        }
+    }
+
+    public double findMinKey() {
+
+        if (min == null) {
+            return -1.0;
+        }
+
+        return min.getKey();
+    }
+
+    public int findMinIndex() {
+
+        if (min == null) {
+            return -1;
+
+        }
+
+        return min.getIndex();
+    }
+
+    public void deleteMin() {
+        if (min == null) {
+            return;
+        }
+
+        FibonacciNode it;
+
```

```
+        n--;
+        //Add childs to root
+
+        FibonacciNode child = min.getChild();
+
+        while (child != null) {
+            min.deleteChild();
+            min.insertRight(child);
+
+            child = min.getChild();
+        }
+
+        if (min == min.getRight() && min == min.getLeft()) {
+            min = null;
+            return;
+        }
+
+
+        min = min.getRight();
+
+        if (min == null) {
+            return;
+        }
+
+
+        min.deleteLeft();
+
+
+        int size;
+        for (size = 0; (1 << size) <= n; size++);
+
+
+        FibonacciNode[] degree = new FibonacciNode[size + 2];
+
+        for (int i = 0; i < size + 2; i++) {
+            degree[i] = null;
+        }
+
+        degree[min.getDegree()] = min;
+
```

```
+        it = min.getRight();
+
+        int count = 1;
+
+        int limit = min.countViz();
+
+        FibonacciNode[] vec = min.getVector();
+
+        while(count++ < limit) {
+
+            it = vec[count - 1];
+
+            while(degree[it.getDegree()] != null) {
+                FibonacciNode newChild = degree[it.getDegree()];
+                degree[it.getDegree()] = null;
+
+                if (it.getKey() < newChild.getKey()) {
+
+                    newChild.getRight().deleteLeft();
+                    it.addChild(newChild);
+
+                    if (newChild == min) {
+                        min = it;
+
+                        if (min == null) {
+                            return;
+                        }
+                    }
+                } else {
+
+                    FibonacciNode rightNode = it.getRight();
+
+                    rightNode.deleteLeft();
+
+
+                    boolean a = (newChild.getRight() == newChild);
+                    rightNode = newChild.getRight();
+                    newChild.getRight().deleteLeft();
+
+                    newChild.addChild(it);
```

```
+
+                    if(!a)
+                        rightNode.insertLeft(newChild);
+
+                    if (it == min) {
+                        min = newChild;
+
+
+                        if (min == null) {
+                            return;
+                        }
+                    }
+                    it = newChild;
+                }
+
+            }
+            degree[it.getDegree()] = it;
+
+            it = it.getRight();
+        }
+
+        FibonacciNode beg = min;
+
+        it = min.getRight();
+
+        while(it != beg) {
+
+            if (it.getKey() < min.getKey()) {
+                min = it;
+
+
+                if (min == null) {
+                    return;
+                }
+            }
+
+            it = it.getRight();
+        }
+
+    }
```

```
+
+    public int size() {
+        return n;
+    }
+
+    public void debug_print() {
+
+        FibonacciNode it;
+
+        if (min == null) {
+            return;
+        }
+
+        it = min.getRight();
+        System.out.printf("It: %f %d\t", min.getKey(), min.getIndex
    ());
+
+        while(it != min) {
+            System.out.printf("%f %d\t", it.getKey(), it.getIndex())
    ;
+
+            it = it.getRight();
+        }
+        System.out.printf("\n");
+    }
+}
diff --git a/src/ons/util/FibonacciNode.java b/src/ons/util/
   FibonacciNode.java
new file mode 100644
index 000000000000..7e6a781516d3
--- /dev/null
+++ b/src/ons/util/FibonacciNode.java
@@ -0,0 +1,235 @@
+/*
+ * To change this license header, choose License Headers in Project
   Properties.
+ * To change this template file, choose Tools | Templates
+ * and open the template in the editor.
+ */
+package ons.util;
```

```
+
+/**
+ *
+ * @author rchehab
+ */
+public class FibonacciNode {
+
+    double key;
+
+    int index;
+
+    FibonacciNode left;
+    FibonacciNode right;
+    int degree;
+    FibonacciNode parent;
+    FibonacciNode child;
+
+    public FibonacciNode() {
+        this.key = 0.0;
+        left = this;
+        right = this;
+        degree = 0;
+
+        parent = null;
+        child = null;
+    }
+
+    public FibonacciNode(double key, int index) {
+        this.key = key;
+        this.index = index;
+        left = this;
+        right = this;
+        degree = 0;
+
+        parent = null;
+        child = null;
+    }
+
+    public double getKey() {
+        return key;
```

```
+    }
+
+    public int getIndex() {
+        return index;
+    }
+
+    public int getDegree() {
+        return degree;
+    }
+
+    public FibonacciNode getChild() {
+        return child;
+    }
+
+    public FibonacciNode getParent() {
+        return parent;
+    }
+
+    public FibonacciNode getRight() {
+        return right;
+    }
+
+    public void setRight(FibonacciNode right) {
+        this.right = right;
+    }
+
+    public FibonacciNode getLeft() {
+        return left;
+    }
+
+    public void setLeft(FibonacciNode left) {
+        this.left = left;
+    }
+
+    public void insertRight(FibonacciNode right) {
+
+        FibonacciNode rightNode = this.getRight();
+
+        this.setRight(right);
+        right.setLeft(this);
```

```
+        right.setRight(rightNode);
+        rightNode.setLeft(right);
+    }
+
+    public void insertLeft(FibonacciNode left) {
+
+        FibonacciNode leftNode = this.getLeft();
+
+        this.setLeft(left);
+        left.setRight(this);
+        left.setLeft(leftNode);
+        leftNode.setRight(left);
+
+    }
+
+    public void deleteRight() {
+
+        if (this == this.getRight() && this == this.getLeft()) {
+            return;
+        }
+
+        FibonacciNode rightNode = this.getRight();
+        FibonacciNode rightrightNode = rightNode.getRight();
+
+        this.setRight(rightrightNode);
+        rightrightNode.setLeft(this);
+
+        rightNode.left = rightNode;
+        rightNode.right = rightNode;
+
+    }
+
+    public void deleteLeft() {
+
+        if (this == this.getRight() && this == this.getLeft()) {
+            return;
+        }
+
+
+        FibonacciNode leftNode = this.getLeft();
```

```
+        FibonacciNode leftleftNode = leftNode.getLeft();
+
+        this.setLeft(leftleftNode);
+        leftleftNode.setRight(this);
+
+        leftNode.left = leftNode;
+        leftNode.right = leftNode;
+
+    }
+
+    public void addChild(FibonacciNode child) {
+        degree++;
+
+        child.parent = this;
+
+        if (this.child == null) {
+            this.child = child;
+            return;
+        }
+
+        this.child.insertRight(child);
+    }
+
+    public void deleteChild() {
+
+        if (child == null) {
+            return;
+        }
+
+        degree--;
+
+        FibonacciNode child = this.child;
+
+        child.parent = null;
+
+        if (child.getRight() == child && child.getLeft() == child) {
+            child.left = child;
+            child.right = child;
+            this.child = null;
+            return;
```

```
+        }
+        this.child = child.getRight();
+        this.child.deleteLeft();
+    }
+
+    public int countViz() {
+
+        int resp = 1;
+
+        FibonacciNode it = this.getRight();
+        while(it != this) {
+            resp++;
+            it = it.getRight();
+        }
+        return resp;
+    }
+
+    public FibonacciNode[] getVector() {
+        int size = countViz();
+
+        FibonacciNode[] vec = new FibonacciNode[size];
+
+        FibonacciNode it = this;
+        for (int i = 0; i < size; i++) {
+            vec[i] = it;
+            it = it.getRight();
+        }
+        return vec;
+    }
+
+    public int debug_print(int level) {
+
+        if (level == 6)
+            System.exit(2);
+
+        int cnt = 1;
+
+        for (int i = 0; i < level; i++)  {
+            System.out.printf("\t");
+        }
```

```
+        System.out.printf("It: %d (%f)\n", this.getIndex(), this.
   getKey());
+
+        if (this.child != null) {
+            cnt += child.debug_print(level + 1);
+        }
+
+
+        FibonacciNode it = this.getRight();
+        while(it != this) {
+
+
+            for (int i = 0; i < level; i++) {
+                System.out.printf("\t");
+            }
+            System.out.printf("It: %d (%f)\n", it.getIndex(), it.
   getKey());
+
+            cnt++;
+
+            if (it.child != null) {
+                cnt += it.child.debug_print(level + 1);
+            }
+
+                it = it.getRight();
+        }
+
+        return cnt;
+    }
+
+}
```

A.7 Modificações na branch ResizableArray

```
diff --git a/src/ons/EONLink.java b/src/ons/EONLink.java
index 76389b6c22ff..7467218f2f6e 100644
--- a/src/ons/EONLink.java
+++ b/src/ons/EONLink.java
@@ -138,6 +138,24 @@ public class EONLink extends Link {
         return slotsAvailable;
```

```
     }

+    /**
+     * Retrieves the set of slots available given a minimum size.
+     *
+     * @param requiredSlots the required slots of set
+     * @return the set with first slots available to 'requiredSlots'
+     */
+    public ArrayList<Integer> getSlotsAvailable2(int requiredSlots)
    {//olha todos os espacos disponiveis levando em cosideracao a
    banda de guarda
+        ArrayList<Integer> slotsAvailable = new ArrayList<>();
+        for (int i = 0; i <= numSlots - requiredSlots; i++) {
+            if (this.slots[i] == 0) {
+                if (areSlotsAvaiable(i, i + requiredSlots - 1)) {
+                    slotsAvailable.add(i);
+                }
+            }
+        }
+        return slotsAvailable;
+    }
+
     /**
      * Retrieves the set of slots available in optical grooming
         given a minimum
     * size.
@@ -200,10 +218,13 @@ public class EONLink extends Link {
     * @return the array with first slots available to '
        requiredSlots'
     */
    public int[] getSlotsAvailableToArray(int requiredSlots) {
-        TreeSet<Integer> slotsAvailable = getSlotsAvailable(
    requiredSlots);
+        ArrayList<Integer> slotsAvailable = getSlotsAvailable2(
    requiredSlots);
+
        int[] out = new int[slotsAvailable.size()];
-        for (int i = 0; i < out.length; i++) {
-            out[i] = slotsAvailable.pollFirst();
+
```

```
+         int i = 0;
+         for (Integer n : slotsAvailable) {
+             out[i++] = n;
          }
          return out;
      }
```

A.8 Modificações na branch GreedyApproach

```
diff --git a/src/ons/EONLink.java b/src/ons/EONLink.java
index 76389b6c22ff..e924cd33bc49 100644
--- a/src/ons/EONLink.java
+++ b/src/ons/EONLink.java
@@ -138,6 +138,75 @@ public class EONLink extends Link {
          return slotsAvailable;
      }

+     /**
+      * Retrieves the set of slots available given a minimum size.
+      *
+      * @param requiredSlots the required slots of set
+      * @return the set with first slots available to 'requiredSlots'
+      */
+     public ArrayList<Integer> getSlotsAvailable2(int requiredSlots)
    {//olha todos os espacos disponiveis levando em cosideracao a
    banda de guarda
+         ArrayList<Integer> slotsAvailable = new ArrayList<>();
+
+         ArrayList<Integer> potentialSlots = new ArrayList<>();
+
+         int cnt = 0;
+         for (int i = 0; i <= numSlots - requiredSlots; i++) {
+             int j;
+             if (this.slots[i] == 0) {
+                 cnt++;
+             } else {
+                 cnt = 0;
+             }
+             if (cnt > requiredSlots) {
+                 cnt = requiredSlots;
```

```
+            }
+            if(cnt == requiredSlots) {
+                potentialSlots.add(i - requiredSlots + 1);
+            }
+        }
+
+        ArrayList<Integer> guardSlots = new ArrayList<>();
+
+        cnt = 0;
+        for (int i = 0; i <= numSlots - requiredSlots; i++) {
+            int j;
+            if (this.slots[i] == 0 || this.slots[i] == -1) {
+                cnt++;
+            } else {
+                cnt = 0;
+            }
+            if (cnt > requiredSlots) {
+                cnt = requiredSlots;
+            }
+            if(cnt == requiredSlots) {
+                guardSlots.add(i - requiredSlots + 1);
+            }
+        }
+
+        int j = 0, k = 0;
+
+        for (int i = 0; i < potentialSlots.size(); i++) {
+
+            int a = potentialSlots.get(i);
+
+            while (j < guardSlots.size() && a >= this.guardband &&
    guardSlots.get(j) + this.guardband < a) j++;
+
+            if (a >= this.guardband && guardSlots.get(j) + this.
    guardband != a) continue;
+
+            if (a >= this.guardband && j >= guardSlots.size()) break
    ;
+
```

```
+            while (k < guardSlots.size() && a <= (numSlots - 1) -
   this.guardband && guardSlots.get(k) < a + this.guardband) k++;
+
+            if (a <= (numSlots - 1) - this.guardband && k >=
   guardSlots.size()) break;
+
+            if (a <= (numSlots - 1) - this.guardband && guardSlots.
   get(k) != a + this.guardband) continue;
+
+            slotsAvailable.add(a);
+        }
+
+        return slotsAvailable;
+    }
+
     /**
      * Retrieves the set of slots available in optical grooming
         given a minimum
     * size.
@@ -200,10 +269,11 @@ public class EONLink extends Link {
     * @return the array with first slots available to '
        requiredSlots'
     */
    public int[] getSlotsAvailableToArray(int requiredSlots) {
-        TreeSet<Integer> slotsAvailable = getSlotsAvailable(
   requiredSlots);
+        ArrayList<Integer> slotsAvailable = getSlotsAvailable2(
   requiredSlots);
        int[] out = new int[slotsAvailable.size()];
-        for (int i = 0; i < out.length; i++) {
-            out[i] = slotsAvailable.pollFirst();
+        int i = 0;
+        for (Integer n : slotsAvailable) {
+            out[i++] = n;
        }
        return out;
    }
```