UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

Paula Franca Toledo

OPERADOR ADAPTATIVO DE MEMÓRIA PARA METAHEURÍSTICAS DE BUSCA LOCAL

Ouro Preto, MG
2021

Paula Franca Toledo

UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profª. Dra. Dayanne Gouveia Coelho

Ouro Preto, MG
2021

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Paula Franca Toledo

Operador Adaptativo de Memória para Metaheurísticas de Busca Local

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 16 de Dezembro de 2021.

Membros da banca

Dayanne Gouveia Coelho (Orientadora) - Doutora - Universidade Federal de Ouro Preto
Rodrigo Geraldo Ribeiro (Examinador) - Doutor - Universidade Federal de Ouro Preto
Pedro Henrique Lopes Silva (Examinador) - Mestre - Universidade Federal de Ouro Preto

Guilherme Tavares de Assis, Coordenador da disciplina "Monografia II" (BCC393) em 2021/1, em nome da Orientadora Dayanne Gouveia Coelho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 16/12/2021.

Documento assinado eletronicamente por **Guilherme Tavares de Assis**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 30/12/2021, às 14:39, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0263083** e o código CRC **6DE0481F**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.013468/2021-94 SEI nº 0263083

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus pais que sempre me incentivaram a estudar e não mediram esforços para que isso acontecesse.

Agradecimentos

Gostaria de agradecer aos meus pais, Ana Maria e Clodoaldo, por terem me apoiado todos os dias, sempre dando amor, companheirismo, conselhos, força e incentivo para que eu seguisse minha caminhada.

Gostaria de agradecer também à minha irmã, Júlia, que sempre esteve ao meu lado, sempre acreditou no meu potencial e me incentivou a vida inteira a ir atrás dos meus sonhos.

Agradeço meu namorado Naywan por todo o apoio, carinho, conselhos dados ao longo do tempo e por ter acreditado que eu conseguiria alcançar esse sonho.

Agradeço também às minhas famílias Franca e Toledo por todo o carinho e incentivo.

À minha orientadora Dayanne Coelho, só tenho a agradecer por tudo que fez por mim e pelo meu aprendizado, por ter sido uma companheira que estava sempre disponível para me ouvir e dar conselhos sobre tudo.

Aos demais professores do DECOM deixo meu agradecimento e levarei os ensinamentos para toda a vida.

Para finalizar, agradeço aos amigos criados ao longo dessa caminhada, especialmente aos remanecentes do 17.2 que sempre estiveram dispostos a me ajudar e fizeram com que a caminhada fosse mais leve.

"Ninguém caminha sem aprender a caminhar, sem aprender a fazer o caminho caminhando, refazendo e retocando o sonho pelo qual se pôs a caminhar."

(FREIRE, 1987)

Resumo

Este trabalho tem por objetivo estudar e propor um operador de vizinhança adaptativo baseado na ideia de memória para metaheurísticas de busca local, com o intuito de aplicar esta técnica desenvolvida na resolução de problemas de otimização combinatória. A ideia principal deste operador é usar a estrutura de memória para evitar revisitações de soluções e direcionar o algoritmo, com base nas informações armazenadas, na realização de uma exploração mais adequada do espaço de busca. O operador descrito será testado na heurística *Simulated Annealing* e na metaheurística *Variable Neighbourhood Search* (VNS). Para investigar a eficiência da metodologia, utilizou-se o Problema de Agendamento de Tarefas com Data de Entrega Comum em uma Única Máquina e o Problema do Caixeiro Viajante. No Problema de Agendamento de Tarefas com Data de Entrega em Comum o VNS-Oadp obteve soluções melhores para a oito instâncias enquanto o VNS-MH apresentou soluções melhores para sete instâncias. Para o problema do Caixeiro Viajante o VNS-Oadp obteve resultados melhores para dez instâncias e o VNS-MH para apenas cinco instâncias. No geral, o operador proposto, melhorou a qualidade das soluções obtidas quando comparados a versão clássica do VNS. Em relação as variações do *Simulated Annealing*, a melhor versão indicada pelos testes para o problema Agendamento de Tarefas com Data de Entrega em Comum foi a SA-Oadp que apresentou sete soluções melhores quando comparado ao SA-MH e se mostrou melhor que o SA clássico em dez problemas. Para o problema do Caixeiro Viajante, a melhor versão indicada pelos testes foi o SA-Oadp que obteve sete soluções melhores que os outros dois métodos. Nos testes realizados, o método proposto se mostrou com um comportamento mais eficiente que as versões originais dos algoritmos, uma vez que foi possível obter melhores soluções médias para algumas instâncias do problema teste com um tempo computacional relativamente curto. Porém, vale destacar que é ainda é necessário fazer uma experimentação estatística mais elaborada para compreender melhor os resultados alcançados e investir numa melhoria mais eficiente e significativa do operador proposto.

Palavras-chave: Metaheurística de busca local; Operadores adaptativos; Estratégias de memória.

Abstract

This work aims to study and propose an adaptive neighborhood operator based on the memory idea for local search metaheuristics, in order to apply this technique developed to solve combinatorial optimization problems. The main idea of this operator is to use the memory structure to avoid revisiting solutions and direct the algorithm, based on the stored information, to carry out a more adequate exploration of the search space. The described operator will be tested in the Simulated Annealing heuristic and the *Variable Neighborhood Search* (VNS) metaheuristic. To investigate the efficiency of the methodology, the Problem of Scheduling Tasks with a Common Delivery Date in a Single Machine and the Traveling Salesman Problem were used. In Task Scheduling Problem with Common Due Date VNS-Oadp got better solutions for eight instances while VNS-MH had better solutions for seven instances. For the Traveling Salesman problem, VNS-Oadp obtained better results for ten instances and VNS-MH for only five instances. Overall, the proposed operator improved the quality of the solutions obtained when compared to the classic version of VNS. Regarding the *Simulated Annealing* variations, the best version indicated by the tests for the Task Scheduling with Delivery Date problem in common was SA-Oadp, which presented seven better solutions when compared to SA-MH and proved to be better than the classic SA in ten problems. For the traveling salesman problem, the best version indicated by the tests was the SA-Oadp, which obtained seven better solutions than the other two methods. In the tests performed, the proposed method showed a more efficient behavior than the original versions of the algorithms, since it was possible to obtain better average solutions for some instances of the test problem with a relatively short computational time. However, it is worth noting that it is still necessary to carry out more elaborate statistical experimentation to better understand the results achieved and invest in a more efficient and significant improvement of the proposed operator.

Keywords: Local search metaheuristics; Adaptive Operators; Memory Strategies.

Lista de Ilustrações

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

ABC	Artificial Bee Colony Algorithm
ADOPT	Adaptive OperaTor Ordering
AG	Algoritmo Genético
AGA	Algoritmo Genético Adaptativo
AP	Adaptative Persuit
BOUX	Based Order Uniform Crossover
BSP	Binary space partitioning
ED	Evolução Diferencial
GRASP	Greedy Randomized Adaptive Search Procedure
GVNS	General Variable Neighborhood Search
ILS	Iterated Local Search
JSP	Job-shop scheduling problem
LOLA	Lower-level algorithm
LS	Local Search
MOHS	Multiobjective Harmony Search
NrGa	Non-Revisiting Genetic Algorithm
NSGA	Non-dominated Sorting Genetic Algorithm
OX	One Point Crossover
PM	Probability Matching
PMX	Partially MappedCrossover
PSO	Particle Swarm Optimization
RRX	Relative Job Order Crossover
SOAD	Self-adaptive operators and alterable search depth
SJOX	Similar Job Order Crossover

TOPD	Problema de Orientação Dependente de Equipe
TOPTW	Problema da Orientação de Equipes com Janelas de Tempo
UPLA	Upper-level algorithm
VND	Variable Neighbourhood Descent
VNS	Variable Neighbourhood Search

Lista de Símbolos

$\mathbb{N}$	Conjunto dos números naturais
$\mathcal{O}$	Limitação superior assintótica
$\in$	Pertence
$\leftarrow$	Atribuição
$\sum$	Somatório (letra grega sigma)
$\subset$	Subconjunto
$\emptyset$	Conjunto vazio
∞	Infinito
α	Letra grega alfa
β	Letra grega beta

Sumário

1 Introdução

Algumas tarefas cotidianas, como por exemplo, encontrar um bom apartamento para morar ou em um supermercado comprar o maior número de produtos com o menor custo possível, buscam determinar o melhor ou o correto dentre um conjunto de possibilidades para se tomar uma ação. Este processo de busca por uma melhor solução é chamado de otimização.

Otimização é uma área da Matemática Aplicada que trata de um conjunto de métodos usados para encontrar uma boa, se possível a melhor, solução para um determinado problema ou que determina as melhores configurações possíveis para a construção ou funcionamento de sistemas de interesse para o ser humano. Geralmente é constituída de três elementos básicos: variáveis de decisão, função-objetivo e restrições.

Nos problemas de otimização, encontrar uma solução para um determinado problema é uma tarefa fácil, porém, encontrar uma boa (ou a melhor) solução pode não ser uma tarefa trivial. A dificuldade em encontrar boas soluções pode ser devido a limitações financeiras, escassez de informações ou até mesmo à natureza conflitante das funções que estão envolvidas no problema.

Existem muitos métodos de otimização e a escolha do método ideal vai depender de uma série de características do problema a ser resolvido e do comportamento deste método quando aplicado ao problema. Em problemas com características difíceis, para os quais não se encontram disponíveis algoritmos determinísticos eficientes, o custo computacional envolvido no processo de otimização é um fator que precisa ser considerado. Por serem facilmente adaptadas a qualquer tipo de problema e fornecer uma solução de boa qualidade em um tempo computacional relativamente aceitável, as metaheurísticas tornaram-se importantes ferramentas para o tratamento de problemas com esse perfil.

Porém, apesar de se mostrarem eficientes, as metaheurísticas apresentam como desvantagem, em relação aos métodos exatos, o fato de não garantirem que a solução final encontrada seja a solução ótima do problema, visto que em alguns casos, o algoritmo pode ser interrompido antes de obter uma solução aceitável.

Para se ter uma boa convergência das metaheurísticas é preciso levar em consideração os seguintes fatores: uma solução inicial de boa qualidade, valores dos parâmetros do algoritmo apropriados, estruturas de vizinhanças adequadas, os operadores específicos e um bom conhecimento das características do problema a ser resolvido. A dependência existente entre o desempenho das metaheurísticas em relação aos fatores citados faz com que muitos pesquisadores concentrem esforços para melhorar estas ações (COELHO, 2016).

Vários estudos propõem o desenvolvimento de novas estratégias e adaptações para as metaheurísticas, com o propósito de aperfeiçoar o desempenho e fazer com que essas ferramen-

tas possam ser usadas para resolver diversos outros problemas. O conceito de adaptação em metaheurísticas parte do princípio de desenvolver operadores que podem sofrer modificações em seu comportamento ao longo da execução do algoritmo. Fialho (2010) define adaptação nos algoritmos como alterações nas ações dos operadores ou nos ajustes de seus parâmetros e, seguindo a mesma ideia, Kramer (2010) utiliza o termo auto-adaptação para definir um sistema que seja capaz de adaptar-se de forma autônoma.

Na Computação Evolutiva, os termos adaptação e auto-adaptação são utilizados, principalmente, para

> denotar técnicas de ajustes ou controle de parâmetros, como por exemplo, o ajuste do tamanho do passo do operador de mutação. A auto-adaptação também é usada durante a execução dos algoritmos para fazer o controle das suas propriedades, tais como os operadores genéticos, os valores dos parâmetros destes operadores ou o tamanho da população (COELHO, 2016).

Neste sentido, Coelho (2016) propõe um operador adaptativo, baseado em uma estratégia de memória, para a metaheurística de busca local *Variable Neighbourhood Search* (VNS). O método utiliza uma estrutura de memória (tabela *hash*) para evitar reavaliações de soluções já visitadas e, um operador de vizinhança adaptativo, para aumentar a exploração no espaço de busca, garantindo um melhor desempenho da metaheurística em relação a convergência e exploração do espaço de soluções.

Visando alcançar um desempenho mais satisfatório das metaheurísticas de busca local, este trabalho tem por objetivo estudar o impacto de um operador adaptativo de busca local com arquivo de memória para a resolução de problemas permutacionais. A ideia principal do operador é orientar o algoritmo com base nas informações armazenadas na estrutura de memória no processo de exploração do espaço de soluções.

Neste contexto, busca-se com esta proposta dar continuidade aos estudos iniciais realizados por Coelho (2016), propondo melhorias e novas estratégias de adaptação ao operador proposto pela autora. Além disso, pretende-se também estudar o impacto do operador proposto quando aplicado na metaheurística VNS e também na heurística *Simulated Annealing*. Para investigar a eficiência da metodologia proposta serão utilizados dois problemas: Problema de Agendamento de Tarefas com Data de Entrega Comum em uma Única Máquina e o Problema do Caixeiro Viajante.

Os resultados iniciais mostram que o método proposto se mostra mais eficiente que as versões originais das metaheurísticas, uma vez que ele obtém melhores soluções médias para a maioria das instâncias com um tempo relativamente curto. Porém, é importante destacar que ainda é necessário fazer um experimentação estatística mais elaborada para compreender melhor os resultados.

1.1 Justificativa

Na literatura, novas técnicas são desenvolvidas com o intuito de melhorar a exploração do espaço factível de soluções, buscando por regiões que sejam mais promissoras neste espaço, com o objetivo de garantir uma convergência mais rápida do algoritmo. Uma região promissora é definida como um subconjunto do espaço de busca que contenha uma quantidade significativa de soluções de boa qualidade para o problema (COELHO, 2016).

Nos trabalhos de Yuen e Chow (2007a), Coelho (2016) são propostos operadores com o objetivo de impedir a reavaliação de uma mesma solução. Esses operadores fazem uso de uma estratégia de memória para armazenar todas as soluções geradas pelos algoritmos e a partir do histórico de soluções, verificar se uma nova solução foi ou não avaliada. Especificamente em (COELHO, 2016), além de evitar a reavaliação esta estrutura é usada pelo operador adaptativo de busca local para orientar na exploração do espaço de busca do algoritmo.

Yuen e Chow (2007a) destaca que a redução da quantidade de avaliações da função e uma boa orientação na exploração do espaço de busca, pode reduzir o esforço computacional e garantir uma convergência mais rápida do algoritmo.

Neste contexto, o estudo por operadores adaptativos de busca local com uma estrutura de memória vinculadas às metaheurísticas justifica-se visto que, o auxílio de uma estratégia de memória permite orientar os operadores de busca local na exploração do espaço de busca e na redução do número de avaliações de funções. Ambos os fatos podem contribuir para uma efetiva aplicação de uma classe de algoritmos para tratar diversos problemas de otimização.

1.2 Objetivos Geral e Específicos

Este trabalho tem como objetivo geral desenvolver um operador adaptativo de busca local com um arquivo de memória para metaheurísticas de busca local. São objetivos específicos:

- Proposição de uma atualização no operador de busca local adaptativo para problemas permutacionais proposto em Coelho (2016);
- Construção de uma nova variante da metaheurística de busca local VNS utilizando o operador de busca local adaptativo;
- Construção de uma nova variante da heurística de busca local *Simulated Annealing* utilizando o operador de busca local adaptativo.

1.3 Método de Trabalho

Visando o alcance do objetivo geral deste trabalho foram realizadas as seguintes ações:

(i) Revisão da literatura fazendo um estudo sobre os conceitos preliminares necessários para o entendimento deste trabalho.

(ii) Pesquisa no estado da arte sobre o funcionamento das metaheurísticas, operadores adaptativos e algoritmos de busca local.

(iii) Estudo e implementação da estrutura de memória desenvolvida no trabalho de Iniciação Científica de Amaro (2019) para o operador proposto.

(iv) Elaboração e implementação do código fonte das metaheurísticas e operadores adaptados aos problemas teste escolhidos.

(v) Experimentos computacionais com a execução do código fonte sobre os conjuntos de instâncias dos problemas-teste.

(vi) Análise e discussão dos resultados alcançados.

1.4 Organização do Trabalho

Este trabalho está dividido em cinco capítulos e se organizado da seguinte maneira: O Capítulo 2 apresenta a fundamentação teórica e um breve resumo dos trabalhos relacionados a proposta desta monografia. O Capítulo 3 detalha o desenvolvimento do operador proposto, sua aplicação em duas metaheurísticas clássicas e como se deu as implementações realizadas.

No Capítulo 4 são apresentados e discutidos os resultados iniciais obtidos pela metodologia proposta e, por fim, no Capítulo 5 são apresentadas as considerações finais e as propostas de continuidade deste estudo.

2 Revisão Bibliográfica

Este capítulo apresenta a revisão bibliográfica acerca do tema proposto e uma breve descrição dos métodos utilizados neste trabalho. Os trabalhos relacionados são apresentados na Seção 2.2 e, na Seção 2.1, a fundamentação teórica necessária para uma melhor compreensão do estudo realizado neste trabalho.

2.1 Fundamentação Teórica

Nesta seção são apresentadas as principais definições acerca de um problema de otimização (Subseção 2.1.1), as heurísticas e metaheurísticas utilizadas no trabalho (Subseção 2.1.2) e os tipos de estruturas de memória que utilizadas em operadores adaptativos (Subseção 2.1.3).

2.1.1 Definições Iniciais

A missão de encontrar a solução ótima para um problema de otimização que envolve apenas uma função-objetivo é chamada de **Otimização Mono-objetivo** ou **Otimização Escalar**. Neste tipo de problema, a função de mérito que se pretende minimizar (maximizar) é uma função cuja imagem é escalar, ou seja, uma função do tipo

$$f : \mathbb{R}^n \rightarrow \mathbb{R}.$$

Nos problemas de otimização, além da minimização (ou maximização) da função objetivo, também é necessário atender algumas restrições. Estas, correspondem ao fato de certas soluções não serem aceitáveis por algum motivo mais subjetivo, ou mesmo por limitações de natureza física ou tecnológica do problema. De forma geral, um problema de otimização mono-objetivo de minimização pode ser expresso por:

$$x^* = \quad \arg\min_x f(x) \tag{2.1}$$

$$\text{sujeito a : } \begin{cases} g_i(x) \leq 0 & \forall i = 1, ..., r \\ h_j(x) = 0 & \forall i = 1, ..., p \end{cases} \tag{2.2}$$

onde x^* é o vetor de variáveis de otimização e $f(x)$ é a função objetivo que se deseja minimizar. A Equação (2.1) significa que é necessário encontrar o vetor $x^* \in \mathbb{R}^n$ que minimize o valor da função $f(x)$. As equações de desigualdades ($g_i(x)$) e de igualdades ($h_i(x)$) da Equação (2.2), representam as restrições que o problema está submetido. Uma solução para este problema, deve satisfazer, sincronicamente, todas as restrições.

A função objetivo pode ser classificada em linear ou não-linear. No caso da otimização linear, interesse deste trabalho, existe uma classe de problemas de otimização em conjuntos finitos chamados de **Otimização Combinatória.** Os problemas de Otimização Combinatória são descritos por estudar arranjos, ordenações ou permutações de um conjunto finito de elementos discretos, que possuem estruturas particulares. Nesses problemas uma solução é representada pela permutação de (n) elementos e encontrar a solução ótima consiste em determinar a melhor solução dentre as $n!$ soluções possíveis do problema.

Nem todos os problemas de otimização combinatória podem ser resolvidos por algoritmos exatos, pois à medida que o valor de n cresce, o tempo computacional para calcular estas soluções também cresce e de forma exponencial. Por não existirem algoritmos que encontrem a solução ótima desses problemas em tempo polinomial eles são classificados no estado da arte como $\mathcal{NP}$-Difíceis. Esse fato faz com que, cada vez mais, métodos baseados em heurísticas e metaheurísticas sejam utilizados para solucionar os problemas de natureza combinatória.

2.1.2 Heurísticas e Metaheurísticas

Souza (2009) define heurística como uma técnica inspirada em processos intuitivos para a busca de uma boa solução com um custo computacional aceitável, sem, no entanto, garantir que a solução encontrada seja a solução ótima do problema ou garantir quão próximo ela está da solução ótima. E Gilovich, Griffin e Kahneman (2002) reforçam que as heurísticas são como atalhos mentais que permitem às pessoas tomarem boas decisões de forma rápida, ajudando a encontrar soluções boas e rápidas para problemas que ainda não possuem a solução ótima.

Glover e Kochenberger (2003) definem metaheurísticas como métodos heurísticos que incluem estratégias melhores para impedir o algoritmo de ficar aprisionado em ótimos locais. As metaheurísticas se diferem de heurísticas convencionais por

> apresentarem um mecanismo que impede que o algoritmo fique preso a um mínimo local. As metaheurísticas podem ser divididas em duas categorias: (i) as que realizam uma busca local e (ii) as que realizam uma busca populacional. Nas metaheurísticas que realizam uma busca local, a exploração no espaço de soluções é feito através de movimentos, em cada iteração do algoritmo, permitindo encontrar uma boa solução na vizinhança da solução analisada. Busca Tabu, *Simulated Annealing* e GRASP são exemplos destas metaheurísticas. As metaheurísticas que realizam uma busca populacional permitem que se explore várias regiões do espaço de soluções a cada iteração do algoritmo, como é feito, por exemplo, nos Algoritmos Genéticos (COELHO, 2011).

Nas subseções seguintes são apresentadas as heurísticas e metaheurísticas utilizadas neste trabalho. A Subseção 2.1.2.1 apresenta a heurística *Variable Neighbourhood Descent* (VND) e as Subseções 2.1.2.2 e 2.1.2.3 apresentam, respectivamente, as metaheurísticas *Variable Neighbourhood Search* (VNS) e *Simulated Annealing*.

2.1.2.1 *Variable Neighbourhood Descent*

O *Variable Neighbourhood Descent* (VND), em português Descida em Vizinhança Variável, é uma heurística de busca local, que aplica várias estruturas de vizinhança, ou seja, utiliza vários movimentos para melhoraria da solução inicial. O conceito dessa heurística é utilizar a mesma estrutura de vizinhança enquanto ela estiver melhorando a solução e, caso isso não esteja mais ocorrendo, avança-se para a estrutura de vizinhança seguinte para garantir melhores soluções. A condição de parada do VND é quando uma solução não pode mais ser melhorada por nenhuma estrutura de vizinhança. O Algoritmo 2.1 apresenta o pseudocódigo do VND conforme apresentado em (GLOVER; KOCHENBERGER, 2003).

Algoritmo 2.1: Variable Neighbourhood Descent

```
   Entrada: N^(.), k_max, f(.), s_i
1  início
2      s* ← s_i
3      k ← 1
4      faça
5          s ← t ∈ N^(k)(s*)
6          se f(s) < f(s*) então
7              s* ← s
8              k ← 1
9          senão
10             k ← k + 1
11     enquanto k ≤ k_max;
12     retorna s
```

No Algoritmo 2.1, os parâmetros de entradas são a vizinhança ($N^{(.)}$), o número máximo de vizinhanças (k_{max}), a função objetivo ($f(.)$) e uma solução inicial (s_i). Na linha 2 do algoritmo atualiza-se a solução corrente do algoritmo. Na linha 5 obtém-se um vizinho aleatório da solução corrente s^*, ao aplicar o movimento da estrutura de vizinhança k. Nas linhas 6 até 8 verifica-se se a função objetivo da solução (s) é menor que o valor da função objetivo da solução corrente, e caso seja verdadeiro, atualiza-se a nova solução corrente e o valor de k, para $k = 1$. Caso a condição apresentada na Linha 6 seja falsa, realiza-se na linha 10 o incremento de k. Enquanto o critério de parada ($k \leq k_{max}$) não for satisfeito o processo da linha 4 à linha 11 será iterado, até que se retorne, no final do algoritmo (Linha 12), a melhor solução s encontrada.

2.1.2.2 *Variable Neighbourhood Search*

O *Variable Neighbourhood Search* (VNS), em português Busca em Vizinhança Variável, é uma metaheurística que aplica várias estruturas de vizinhança para percorrer uma heurística de

busca local. O Algoritmo 2.2 mostra o pseudocódigo desta metaheurística.

Algoritmo 2.2: Variable Neighbourhood Search

```
   Entrada: N^(.), k_max, f(.), s_i
1  início
2      s* ← s_i
3      faça
4          k ← 1
5          faça
6              s' ← t ∈ N^(k)(s*)
7              s ← buscaLocal(s')
8              se f(s) < f(s*) então
9                  s* ← s
10                 k ← 1
11             senão
12                 k ← k + 1
13         enquanto k ≤ k_max;
14     enquanto condição de parada não é satisfeita;
15     retorna s
```

Os parâmetros de entradas são a vizinhança ($N^{(.)}$), o número máximo de vizinhanças (k_{max}), a função objetivo ($f(.)$) e uma solução inicial (s_i). O VNS parte da solução inicial, ou seja, a solução corrente s^* = s_i e, enquanto não atingir o critério de parada, os seguintes passos são repetidos: inicializa a estrutura de vizinhança $k = 1$ (linha 4) e um vizinho aleatório s' de s^* é selecionado considerando o movimento k (linha 6); na linha 7 é empregada uma busca local neste vizinho e verifica-se através do critério de aceitação (linha 8) verifica-se se a nova solução é melhor que a solução corrente. Se a solução resultante da busca local, s, for melhor que a solução corrente s^*, esta é atualizada (linhas 9 e 10). Caso a busca local fique presa em um mínimo local, ou seja, não produz nenhuma solução melhor, o algoritmo altera a estrutura de vizinhança por outra que ainda não foi utilizada (linha 12). O processo executa uma nova busca local na solução atual, usando uma nova vizinhança e este processo é repetido até que o critério de parada seja satisfeito. A condição de parada do VNS não é originalmente definida, podendo ser feita considerando o número máximo de iterações, percentual de melhora, tempo, ou outro fator dependendo do problema a ser tratado. Maiores detalhes a respeito do funcionamento da metaheurística pode ser obtido em Glover e Kochenberger (2003).

Na literatura, é comum chamar de *General Variable Neighborhood Search* (GVNS) a metaheurística VNS utilizando o VND como heurística de busca local. Outras variações do VNS também podem ser obtidos em Glover e Kochenberger (2003).

2.1.2.3 *Simulated Annealing*

A heurística *Simulated Annealing* (SA), que é traduzido em português como Recozimento Simulado, tem esse nome pois simula um processo metalúrgico chamado de recozimento, que consiste em deixar uma placa ou liga metálica na sua forma de treliça mais regular possível. Isso é possível esquentando-se a placa e submetendo-a a um resfriamento suficientemente lento. A metaheurística realiza um procedimento parecido com o espaço de busca de um problema de otimização combinatória: vizinhos da solução corrente com melhor valor de função são sempre aceitos e, os vizinhos com valor de função igual ou pior poderão ser aceitos conforme uma certa probabilidade, que começa alta e vai diminuindo a cada iteração até que se atinja uma boa solução, da mesma forma da temperatura do processo metalúrgico.

O Algoritmo 2.3 descreve o funcionamento da heurística conforme proposto em (GLOVER; KOCHENBERGER, 2003).

Algoritmo 2.3: Simulated Annealing

Entrada: $N(.)$, $f(.)$, s, c, T
Saída: s

```
1  início
2      t_k ← T
3      faça
4          faça
5              s' ← w ∈ N(s)
6              Δ_{s,s'} ← f(s') − f(s)
7              se Δ_{s,s'} ≤ 0 então
8                  s ← s'
9              senão
10                 s ← s' com probabilidade exp(−Δ_{s,s'}/t_k)
11         enquanto condição de parada não é satisfeita;
12         t_k ← t_k * (1 − c)
13     enquanto t_k ≥ 0;
14 retorna s
```

Os parâmetros de entradas são a vizinhança ($N(.)$), a função objetivo ($f(.)$), a taxa de resfriamento c e a temperatura inicial T. Na linha 2 a variável t_k recebe a temperatura inicial passada por parâmetro. Na linha 5 s' recebe um vizinho aleatório de s; na linha 6 $\Delta_{s,s'}$ recebe a diferença do valor da solução nova com o valor da solução original e caso essa diferença seja menor ou igual a zero (linha 7) a solução nova é atribuída à solução original (linha 8). Caso contrário a solução nova é atribuída a solução original com uma probabilidade que é calculada por exponencial de $(-\Delta_{s,s'}/t_k)$ (linha 10). O processo da linha 3 até a linha 10 é repetido enquanto uma condição de parada não for satisfeita. Após a satisfação dessa condição a variável t_k recebe

o seu valor multiplicado por 1 menos a taxa de resfriamento (linha 12). O processo da linha 3 até a linha 13 é repetido até que t_k seja menor ou igual a 0.

2.1.3 Estruturas de Memória

Uma estrutura de memória pode ser compreendida como uma conjunto de elementos que ficam guardados na memória principal ou secundária do computador, sob alguma política de organização que atende à alguma finalidade. As estruturas de memória mais utilizadas em conjunto com as heurísticas/metaheurísticas são brevemente apresentadas nas subseções seguintes. Maiores detalhes a respeito da complexidade e implementação destas estruturas de memória podem ser obtidas em (ZIVIANI, 2007; FUCHS; KEDEM; NAYLOR, 1980)

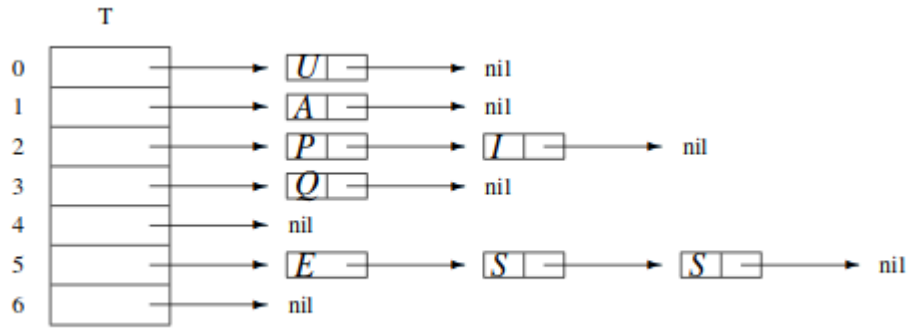
2.1.3.1 Tabela *Hash*

A tabela *hash* é uma estrutura de dados frequentemente representada por uma matriz, de $i \in \mathbb{N}$ posições indexadas. Todo elemento adicionado à tabela *hash* deve ter uma ou mais chaves, que servirão de entrada para uma função **hash** que produzirá como saída a posição i que o elemento será ser adicionado.

O custo computacional pela inserção e busca por um elemento pela chave é constante e expressado por $\mathcal{O}(1)$. Num mundo ideal, a função deveria prover uma correspondência um-a-um entre os possíveis valores de chave e as posições da matriz. Porém, elaborar essa função de anseio pode ser muito difícil ou impossível, razão pela qual as funções *hash* geralmente são passíveis de ocasionarem **colisões**. As colisão são quando dois ou mais elementos distintos são inseridos em uma mesma posição. Para evitar isto podem ser implementadas listas encadeadas nas posições em que dois ou mais elementos são inseridos.

Com a implementação dessas listas encadeadas é importante destacar que quando vários elementos são inseridos em uma mesma posição a busca por um elemento não poderá mais ser realizada em tempo computacional constante. A complexidade da busca por um registro na tabela *hash* decorrerá da qualidade da função *hash* escolhida.

Figura 2.1 – Exemplo de uma tabela *hash*.



Fonte: (ZIVIANI, 2007).

A Figura 2.1 ilustra essa o funcionamento da estrutura de dados tabela *hash*. Há uma lista T de posições indexadas em ordem crescente, de 0 até 6, onde nenhum elemento inserido teve 4 ou 6 como resposta da função hash, e as demais posições estão preenchidas. Todas as posições possuem uma lista, para evitar a colisão, e vemos que as posições 2 e 5 já tiveram colisão.

No trabalho de (COELHO, 2016) é utilizado essa estrutura para armazenar as soluções já calculadas e não ter que recalcular o valor da solução, pois a função objetivo pode ser muito custosa. No presente trabalho também é utilizado a tabela *hash* com o mesmo objetivo do trabalho da Coelho (2016).

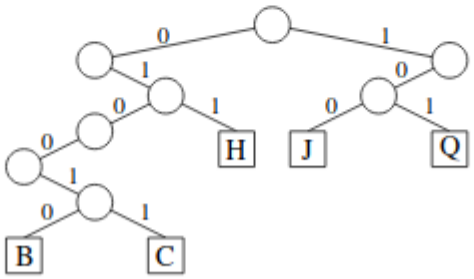
2.1.3.2 Árvore de Prefixos

A estrutura de dados Árvore de Prefixos (ou, *Trie*), é uma árvore M-ária onde cada elemento é adicionado conforme a ordem dos componentes que formam sua chave.

Num nível i qualquer da árvore, todos os elementos possuem o mesmo prefixo. Uma característica interessante dessa estrutura é que podemos codificar as chaves em binário, o que implica que a árvore também será binária. Existem algoritmos que exploram essa propriedade para fornecer bons tempos de inserção e busca, como o algoritmo PATRICIA.

Figura 2.2 – Árvore de prefixos binária.

$$
\begin{aligned}
\mathrm{B} &= 010010 \\
\mathrm{C} &= 010011 \\
\mathrm{H} &= 011000 \\
\mathrm{J} &= 100001 \\
\mathrm{M} &= 101000
\end{aligned}
$$



Fonte: (ZIVIANI, 2007)

A Figura 2.2 ilustra uma árvore de prefixos com algumas chaves codificadas em binário. Primeiramente a imagem mostra as codificações binárias dos elementos inseridos, que são as

letras B, C, H, J e M. Neste caso em particular, qualquer ordem de inserção produz a mesma árvore na parte inferior da figura, onde os nós-folhas são os elementos inseridos, e os nós intermediários são as cadeias binárias da codificação. Elementos sob o mesmo nó-raiz em cada subárvore possuem um mesmo prefixo, como por exemplo as letras J (100) e Q (101).

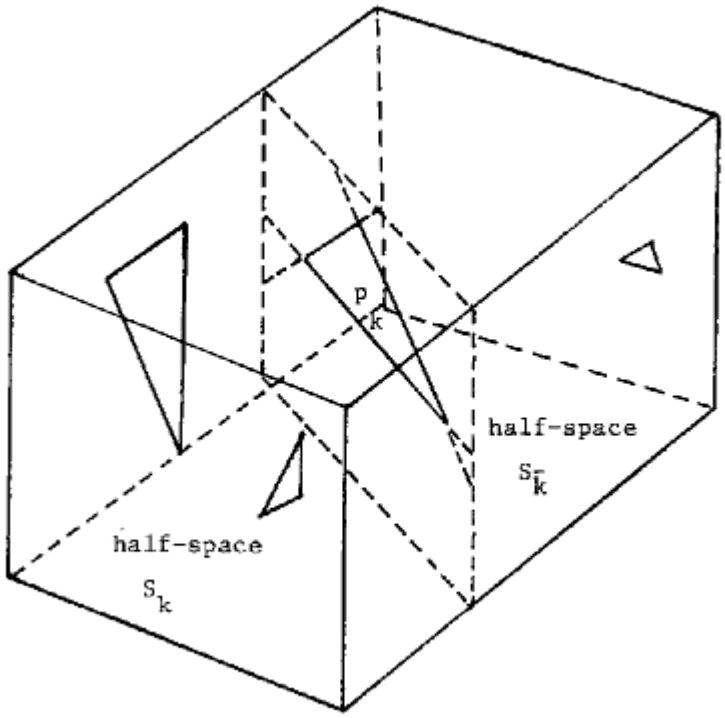
Šramko (2009) propõe o uso da estrutura de **árvore de prefixos**, para guardar soluções de problemas combinatórios e evitar/impedir revisitações e a mesma apresentou-se vantajosa por sempre direcionar o algoritmo genético na melhoraria das soluções obtidas permitindo uma boa exploração do espaço de busca.

2.1.3.3 Árvore de Particionamento Binário de Espaço

A estrutura de dados Árvore de Particionamento Binário de Espaço (*Binary space partitioning* (BSP)) foi proposta por Fuchs, Kedem e Naylor (1980) como uma solução para um problema de computação gráfica, chamado **renderização**. O problema consistia em calcular os pixels a serem transmitidos para um dispositivo de saída de vídeo a partir de um modelo 3D de polígonos. A ideia principal da árvore é dividir o espaço em prioridades de visualização, com respeito a um ponto específico, chamado de **ponto de observação**. Com essa estrutura montada, um algoritmo pode ser executado para determinar quais os polígonos, ou partes de polígonos são visíveis a partir daquele ponto e então calcular os pixels que devem ser passados ao dispositivo de saída de vídeo.

A Figura 2.3 ilustra os particionamentos binários do espaço aplicados em um cenário 3D, que é basicamente o que compõe a árvore. Na figura vemos o cenário com alguns objetos triangulares, e o espaço visual sendo repartido em dois subespaços, onde o espaço S_k é a parte visível para um observador na face menor à esquerda e o outro subespaço é a parte não-visível para este observador (e que portanto não será renderizado).

Figura 2.3 – Árvore de particionamento binário de espaço.



Fonte: (FUCHS; KEDEM; NAYLOR, 1980)

Apesar da natureza da estrutura, é possível aplicá-la em outras situações onde podemos interpretar o espaço do problema como um espaço visual, como por exemplo, no espaço de busca de problemas de otimização combinatória, processo realizado por Yuen e Chow (2007b).

Yuen e Chow (2007b) propõem o uso de uma árvore *Binary space partitioning*(BSP) para garantir a não-revisitação de uma mesma solução em um algoritmo genético.

Neste trabalho foi utilizado a tabela *hash* para armazenar o valor das soluções que já tenham sido calculadas e assim minimizar o tempo gasto para um novo cálculo dessa função objetivo, pois diversos problemas apresentam funções objetivo muito custosas.

Foi escolhido essa estrutura de memória pois o custo computacional pela inserção e busca por um elemento pela chave é constante e expressado por $\mathcal{O}(1)$.

2.2 Trabalhos Relacionados

A palavra adaptativo, na biologia, é definida como a capacidade dos seres vivos de se adaptarem ao ambiente em que vivem. Na computação evolutiva, este termo parte do mesmo princípio e a aplicação deste conceito nos algoritmos evolutivos está relacionada a proposição de técnicas de ajustes ou controle de parâmetros, como por exemplo, o ajuste do operador de mutação.

Como os algoritmos evolutivos apresentam uma convergência mais lenta quando comparados às demais metaheurísticas, muitos trabalhos concentram esforços em desenvolver mecanismos que melhore a eficiência e a convergência destes algoritmos. A maioria dos trabalhos que se propõem a estudar operadores adaptativos, fazem uso desta técnica em Algoritmos Genéticos (AGs), desenvolvendo novos operadores e/ou criando adaptações aos operadores usuais de mutação, cruzamento e seleção.

No trabalho de Gonçalves et al. (2016) são investigadas duas técnicas de seleção adaptativa de operadores do algoritmo *Non-dominated Sorting Genetic Algorithm III* (NSGA-III) que são: (*Adaptive Pursuit* e *Probability Matching*) para escolher em tempo de execução qual o operador é mais eficiente para a resolução do Despacho Econômico de Energia Elétrica. A seleção do operador da Evolução Diferencial (ED) a ser aplicada é estocástica, dependendo apenas da probabilidade atual de aplicação de cada operador, que, inicialmente, possuem a mesma probabilidade. De acordo com a operação selecionada são escolhidos alguns indivíduos para a aplicação do operador.

Em Gonçalves et al. (2017) são propostos dois mecanismos adaptativos para fazer a seleção dos operadores do NSGA-III: o *Probability Matching* (PM) e o *Adaptive Pursuit* (AP). Estes mecanismos são incorporados na estrutura do algoritmo para selecionar de forma autônoma o operador adequado o algoritmo vai utilizar em um problema multi-objetivo. Os mecanismos de seleção de operadores escolhem os operadores para gerar novas soluções com base nas informações coletadas pelos métodos de atribuição de crédito. A seleção do operador a ser aplicado ocorre com base nas probabilidades associadas a cada operador para ambos: NSGA-III AP e NSGA-III PM. À medida que o aumento de uma probabilidade ocorre com base no sucesso de um operador, o operador mais bem sucedido será selecionado com mais frequência, melhorando teoricamente os resultados.

No trabalho de Jiang et al. (2016) é tratado o problema de programação entre células com limitações de capacidade de transporte, que é essencialmente a coordenação da programação de peças e transporte intercelular, por ser um problema de alta complexidade é utilizado o *Artificial Bee Colony* (ABC) (Côlonia Artificial de abelhas) com o operador *self-adaptive operators and alterable search depth*(SOAD), no português operador auto-adaptativo e profundidade de pesquisa alterável. Através da introdução do SOAD na *employed bee phase*, a adoção de operadores genéticos e a profundidade de pesquisa correspondente são adaptáveis determinado de acordo com seu desempenho histórico. O mecanismo auto-adaptativo SOAD é incorporado nas decisões sobre os operadores e sua busca em profundidade. Os operadores com bom desempenho têm maiores probabilidades de serem selecionados, e com base no desempenho de um operador, sua profundidade de pesquisa é determinada dinamicamente por uma árvore de decisão. No trabalho são utilizadas quatro tipos de movimentos: trocas aleatórias, trocas aleatórias de subsequências, inserções aleatórias, inserções aleatórias de subsequências.

Gunawan, Lau e Lu (2018) utilizam a estrutura, chamada *Adaptive Operator Ordering*

(ADOPT) nas metaheurísticas *Iterated Local Search* (ILS) e na hibridização do *Simulated Annealing* com o ILS (SAILS) para resolver duas variantes do Problema de Orientação: o Problema de Orientação Dependente de Equipe, do inglês *Team Dependent Orienteering Problem* (TDOP) e o Problema da Orientação de Equipes com Janelas de Tempo, do inglês *Team Orienteering Problem with Time Windows* (TOPTW). A estrutura ADOPT acomoda um mecanismo de aprendizagem de reforço (baseado em aprendizagem de autômatos) que calcula a probabilidade de selecionar um operador da busca local, do inglês *Local Search* (LS), a fim de calcular a probabilidade de selecionar o melhor operadores para a próxima iteração. Os valores de probabilidade são usados para gerar uma sequência / ordem de operadores para a próxima iteração da busca local, com base em três estratégias de ordenação diferentes: seleções baseadas em classificação, aleatórias e proporcionais de adequação.

Pongchairerks (2020) utiliza para resolver o problema *job-shop scheduling problem* (JSP) uma metaheurística de dois níveis, sendo o nível superior o *upper-level algorithm* (UPLA) que controla os parâmetros de entrada do segundo nível chamado *lower-level algorithm* (LOLA) e sua função é buscar o cronograma ideal.

O LOSAP (um algoritmo de busca local que seleciona a solução ótima em uma estrutura de vizinhança híbrida baseada em probabilidade), usa aleatoriamente um dos dois operadores vizinhos predeterminados por uma probabilidade pré-atribuída para gerar cada solução vizinha. Um valor alto dessa probabilidade leva a estrutura de vizinhança híbrida a ser mais semelhante à estrutura de vizinhança do primeiro operador e vice-versa. A adaptação deste algoritmo é feita pela probabilidade de escolher um dos dois operadores predeterminados.

Yuen e Chow (2009b), Yuen e Chow (2009a) propõem um operador de mutação adaptativo para o *Non-Revisiting Genetic Algorithm* (NrGa), proposto por Yuen e Chow (2007a). O ajuste do tamanho do passo do operador é realizado conforme a organização das soluções armazenadas na BSP. Esta mutação adaptativa é utilizada para buscar vizinhos que estejam numa sub-região ainda não explorada e que esteja próximo a solução atual. O operador é utilizado somente para regular o tamanho do passo e este tamanho é alterado conforme o tamanho da sua sub-região. O tamanho do passo mostra também se uma região foi pouco visitada e caso ela tenha sido pouco visitada o valor da taxa de mutação é aumentado imediatamente.

Thierens (2002) propõe um operador de mutação adaptativo para controlar a taxa de probabilidade na mutação considerando o valor de aptidão do pai. No operador, os indivíduos apresentam uma probabilidade de mutação baixa se o valor de aptidão do pai for alta. Do contrário, se o valor de aptidão do pai for baixo, aumenta-se a probabilidade de mutação.

(GERALDO; MARCHI, 2012) propõem melhorias nos principais operadores de busca e de diversidade do algoritmo *Multiobjective Harmony Search* (MOHS). Os autores substituem o mecanismo de aleatoriedade do algoritmo original por outro menos disruptivo. O operador de cruzamento original, por exemplo, é substituído pelo operador BLX-alfa e a perturbação original por um operador de mutação polinomial. Já em Navas e Urbaneja (2013) é realizado um estudo

acerca do efeito de operadores adaptativos na metaheurísticas *Non-Revisiting Genetic Algorithm II* (NSGA-II). Os autores desenvolveram duas versões do algoritmo: o NSGAIIrd que utiliza os operadores de forma aleatória e o NSGAIIad que utiliza um esquema que calcula a contribuição de cada operador para a busca da solução exata.

Ribeiro, Souza e Souza (2010) propõem a utilização de um algoritmo genético adaptativo para a resolução do Problema de Sequenciamento em uma Máquina com Minimização das Penalidade por Antecipação e Atraso da Produção. O algoritmo proposto é o Algoritmo Genético Adaptativo (AGA) e usa a fase de construção metaheurística *Greedy Randomized Adaptive Search Procedure* (GRASP) para a construção da população inicial. Para a geração de novos indivíduos foram usados cinco diferentes operadores de cruzamento, inicialmente, com a mesma taxa de probabilidade. A adaptação durante o processo evolutivo ocorre fazendo a alterações nos valores de probabilidade de cada operador, o que influencia diretamente sobre qual será aplicado. Esse valor muda de acordo com o desempenho do operador, de forma que os melhores recebem uma probabilidade maior. Os cinco operadores de cruzamento utilizados são o *One Point Crossover* (OX), o *Similar Job Order Crossover* (SJOX), o *Relative Job Order Crossover* (RRX), o *Based Order Uniform Crossover* (BOUX) e o *Partially Mapped Crossover* (PMX).

Na literatura, a aplicação de estruturas de memória em diferentes metaheurísticas não é uma temática muito aplicada. O algoritmo mais conhecido que faz uso de memória é a metaheurística Busca Tabu.

Yuen e Chow (2007b) propõem o uso de uma árvore *Binary space partitioning*(BSP) (veja Subseção 2.1.3.3) para garantir a não-revisitação de uma mesma solução em um algoritmo genético. Os autores também aplicaram a metodologia proposta nos algoritmos de Otimização por Enxame de Partículas, do inglês *Particle Swarm Optimization* (PSO), em um *Simulated Annealing* e em um algoritmo genético para impedir revisitações e acelerar a convergência dos algoritmos (CHOW; YUEN, 2008; YUEN; CHOW, 2008b; YUEN; CHOW, 2009b). A memória também foi aplicada para estimular a exploração de outras regiões do espaço de busca ainda não exploradas pelos algoritmos.

Yuen e Chow (2008a) utiliza um algoritmo genético com a estrutura de memória de uma árvore BSP para tratar o problema do caixeiro viajante. No trabalho, são apresentadas ilustrações acerca do funcionamento da inserção de uma solução na árvore. Chow e Yuen (2011) mostra que é possível apoiar sobre a estrutura da BSP para tomar decisões ao longo das iterações do algoritmo, criando novos operadores adaptativos eficientes que se baseiam em estruturas de memória.

Motivado pelo fato de que a estrutura de memória apresentada por Yuen e Chow (2007b) só funciona bem para problemas contínuos, Šramko (2009) publicou um livro onde propõe o uso da estrutura de **árvore de prefixos** (Subseção 2.1.3.2), em inglês chamada de *Trie*, para guardar soluções de problemas combinatórios e evitar/impedir revisitações. Essa abordagem apresentou-se ruim para problemas onde a descoberta de melhores soluções é feita aleatoriamente

e quando o número de revisitações é baixo comparado ao tamanho do espaço de busca. No entanto, apresentou-se vantajosa por sempre direcionar o algoritmo genético na melhoraria das soluções obtidas permitindo uma boa exploração do espaço de busca, ao forçar maiores taxas de mutação em regiões do espaço de busca que foram visitadas mais frequentemente. Os autores comparam as complexidades em notação assintótica da árvore de prefixos com a tabela *hash* e a árvore binária de pesquisa, para justificar a razão pela qual utilizou-se a árvore de prefixos em primeiro lugar: essa estrutura se mostrou mais apropriada para ser aplicada às metaheurísticas populacionais. Outra diferença com relação à BSP é que o espaço de busca precisa ser tratado em binário.

Fundamentado nos trabalhos de (CARRANO; MOREIRA; TAKAHASHI, 2011) e (YUEN; CHOW, 2007b), Coelho (2016) faz um estudo da aplicação de estruturas de memória em um operador adaptativo para a metaheurística de busca local VNS. O operador proposto possui como objetivo promover uma busca considerando uma vizinhança adaptativa das soluções replicadas pelo algoritmo. A ideia é obter uma nova solução que ainda não foi visitada. A autora considera quatro movimentos nas vizinhanças e utiliza um conceito de blocos para criar vizinhanças ainda não exploradas pelo algoritmo. Os movimentos utilizados foram: troca de ordem entre dois blocos na sequencia adjacente, troca de ordem entre dois blocos quaisquer da solução, realocação de um bloco em outra posição e troca de ordem entre três blocos quaisquer da solução. Outra contribuição relevante do trabalho de (COELHO, 2016) consiste na aplicação de uma nova função *hash* para lidar com problemas de otimização combinatória e que garante ao mesmo tempo um baixo número de colisões. Este trabalho é o que mais se aproxima do estudo proposto nesta monografia.

Neste trabalho foi implementado o operador adaptativo semelhante ao apresentado em (COELHO, 2016). No trabalho de referência o operador adaptativo utiliza 4 estruturas de vizinhança e divide a solução de forma decrescente em n blocos, partindo de $n = tamanhodasolução/2$ e os movimentos nestas estruturas de vizinhanças acontecem por blocos. O presente trabalho, se difere da abordagem proposta em (COELHO, 2016) por utilizar 3 estruturas de vizinhanças e por dividir cada solução d forma crescente em n blocos, partindo de $n = 2$ até $n = tamanhodasolução/2$. Vale destacar que, assim como (COELHO, 2016) o operador de adaptativo só será aplicado as soluções replicadas, ou seja, em soluções que já foram avaliadas e que o algoritmo já determinou o seu melhor vizinho aplicando a busca local convencional.

3 Desenvolvimento

Este capítulo apresenta o desenvolvimento do trabalho proposto, que tem como objetivo desenvolver um operador de busca local adaptativo com estratégia de memória para algoritmos clássicos de busca local. O operador proposto pode ser aplicado em qualquer metaheurística (ou heurística) que utilizam busca local. Neste trabalho são utilizadas a metaheurística Busca em Vizinhança Variável - *Variable Neighborhood Search* e a heurística *Simulated Annealing*. A estratégia de memória usada foi implementada por Amaro (2019).

Neste contexto, são desenvolvidas duas versões para os algoritmos utilizados, que neste trabalho são tratadas nos capítulos e seções seguintes por:

VNS-Oadp: consiste na versão clássica do VNS com operador adaptativo de busca local (Seção 3.3).

SA-Oadp: consiste na versão clássica do SA com operador adaptativo de busca local (Seção 3.4).

O restante do capítulo está organizado da seguinte maneira: a Seção 3.1 descreve os tipos de movimentos e as estruturas de vizinhança utilizadas; a Seção 3.2 apresenta o operador de busca local adaptativo proposto; e as Seções 3.3 e 3.4 apresentam, respectivamente, o VNS-Oadp e o SA-Oadp.

3.1 Estruturas de Vizinhança

Uma solução, em um problema de otimização combinatória, é representada por uma combinação de n números inteiros. Para $n = 8$, uma solução s por exemplo, pode ser representada por $s = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

A vizinhança de uma solução s é o conjunto das soluções $N(s) \in S$ em que S representa o espaço de busca. Toda solução vizinha de s, representada por $s' \in N(s)$, é obtida a partir de um movimento realizado nesta solução. E movimento, por sua vez, é a troca de posições em uma solução, assim com esses movimentos geramos soluções diferentes da solução inicial.

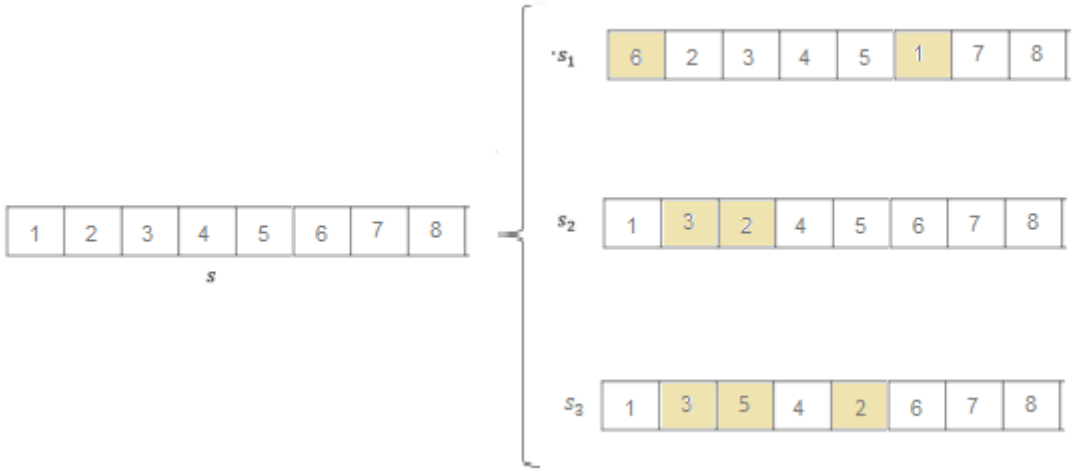
Neste trabalho são realizados três tipos de movimentos que definem as três estruturas de vizinhança utilizadas, são eles:

1. Troca de duas posições aleatórias do vetor solução.
2. Troca de duas posições consecutivas do vetor solução.

3. Troca de três posições aleatórias do vetor solução.

Os movimentos compõem, respectivamente, as vizinhanças $N^{(1)}$, $N^{(2)}$ e $N^{(3)}$. A Figura 3.1 ilustra os três movimentos usados e apresenta um exemplo de cada vizinho gerado a partir de uma solução hipotética s.

Figura 3.1 – Exemplo de movimento realizado na estrutura de vizinhança $N^{(1)}$, $N^{(2)}$ e $N^{(3)}$.



Na figura 3.1, o vizinho S_1 foi obtido aplicando o movimento 1, onde foram escolhidas aleatoriamente as posições 1 e 6 e feito a troca dos elementos destas posições. Já o vizinho S_2 foi obtido aplicando o movimento 2, onde foi escolhida a posição 2 aleatoriamente e posição 3 foi selecionada por ser consecutiva a posição 2 e feito a troca dos elementos destas posições. O vizinho S_3 por sua vez foi obtido aplicando o movimento 3, onde foram escolhidas aleatoriamente as posições 2,3 e 5 e feito a troca dos elementos destas posições.

3.2 Operador Busca Local Adaptativo

O operador de Busca Local é aplicado em toda solução já avaliada e refinada pelo algoritmo. Nesta solução é realizada uma nova busca local utilizado a estrutura de vizinhança adaptativa. Os movimentos na estrutura de vizinhança são realizados por blocos e o tamanho desses blocos é modificado ao longo da execução do algoritmo. O objetivo do uso da vizinhança adaptativa é permitir que algoritmo explore diferentes regiões do espaço de busca que ainda não foram investigadas.

Todas as soluções obtidas são armazenadas na estrutura de memória e, para saber se uma solução foi ou não avaliada, é realizada uma busca nesta estrutura. A estratégia de memória utilizada foi implementada conforme apresentado em Amaro (2019). As implementações estão disponíveis no link <https://github.com/mayconamaro/memoriaMetaheuristicas>.

A seguir são apresentados os algoritmos 3.1 e 3.2 e cada algoritmo utiliza um tipo de movimento.

O algoritmo 3.1 representa os movimentos 1 e 2 que são consecutivamente: troca de dois blocos aleatórios do vetor solução e troca de dois blocos consecutivos do vetor solução.

Algoritmo 3.1: Operador Adaptativo - Troca de dois blocos aleatórios

Entrada: s_i, $tamBloco$
Saída: s

```
1  início
2      qtdeBlocos ← n/tamBloco
3      bloco1 ←recebe um valor aleatório entre 1 e qtdeBlocos
4      faça
5          bloco2 ← recebe um valor aleatório entre 1 e qtdeBlocos
6      enquanto bloco1 == bloco2;
7      aux ← s[bloco1]
8      s[bloco1...bloco1 + tamBloco] ← s[bloco2...bloco2 + tamBloco]
9      s[bloco2...bloco + tamBloco] ← s[bloco1bloco1...bloco1 + tamBloco]
10 retorna s
```

No algoritmo 3.1, as entradas são a solução corrente s o tamanho atual do bloco $tamBloco$. Na linha 2 uma variável recebe o tamanho da solução dividido pelo tamanho do bloco para sabermos quantos blocos teremos na solução. Uma variável chamada bloco1 recebe um número aleatório entre 1 e quantidade de blocos para escolher o primeiro bloco que será trocado (linha 3). Temos um laço de repetição para fazer a certificar que o valor de bloco1 seja diferente de valor do bloco2. Após esse laço temos da linha 7 a linha 9 a troca entre os blocos e a atualização destes blocos na solução. Esse código é para o movimento de trocar dois blocos aleatórios do vetor solução.

Para o movimento troca de dois blocos consecutivos do vetor a mudança acontece na linha 5 do algoritmo 3.1 onde o código passa a ser:

$$bloco2 \leftarrow \text{recebe o valor do} bloco1 + tamBloco$$

Também é retirado o loop das linhas 4 e 6, pois não será necessário mais verificar se os valores de bloco1 e bloco2 são diferentes visto que bloco2 não recebe um valor aleatório e sim um valor baseado no valor do bloco1.

O algoritmo 3.2 representa o movimento 3 que é a troca de três blocos aleatórios do vetor

solução.

Algoritmo 3.2: Operador Adaptativo - Troca de três blocos aleatórios

Entrada: $s_i[1, ..., n]$, $tamBloco$
Saída: s

```
1  início
2      qtdeBlocos ← n/tamBloco
3      bloco1 ←recebe um valor aleatório entre 1 e qtdeBlocos
4      faça
5          se bloco1 == bloco2 então
6              bloco2 ← recebe um valor aleatório entre 1 e qtdeBlocos
7          se bloco2 == bloco3 então
8              bloco3 ← recebe um valor aleatório entre 1 e qtdeBlocos
9          se bloco1 == bloco3 então
10             bloco3 ← recebe um valor aleatório entre 1 e qtdeBlocos
11     enquanto bloco1 == bloco2 || bloco2 == bloco3 || bloco3 == bloco1;
12     s[bloco1...bloco1 + tamBloco] ← s[bloco3...bloco3 + tamBloco]
13     s[bloco2...bloco + tamBloco] ← s[bloco1...bloco1 + tamBloco]
14     s[bloco3...bloco + tamBloco] ← s[bloco2...bloco2 + tamBloco]
15 retorna s
```

Na linha 2 do algoritmo 3.2 uma variável recebe o tamanho da solução dividido pelo tamanho do bloco para sabermos quantos blocos teremos na solução. Uma variável chamada bloco1 recebe um número aleatório entre 1 e quantidade de blocos para escolher o primeiro bloco que será trocado (linha 3). Temos um laço de repetição para fazer a certificar que o valor de bloco1 seja diferente de valor do bloco2 e diferente do valor de bloco3. Após esse laço temos da linha 12 a linha 14 a troca entre os blocos e a atualização destes blocos na solução. Esse código é para o movimento de trocar três blocos aleatórios do vetor solução que é muito parecido com a troca de dois blocos aleatórios do vetor solução do algoritmo 3.1

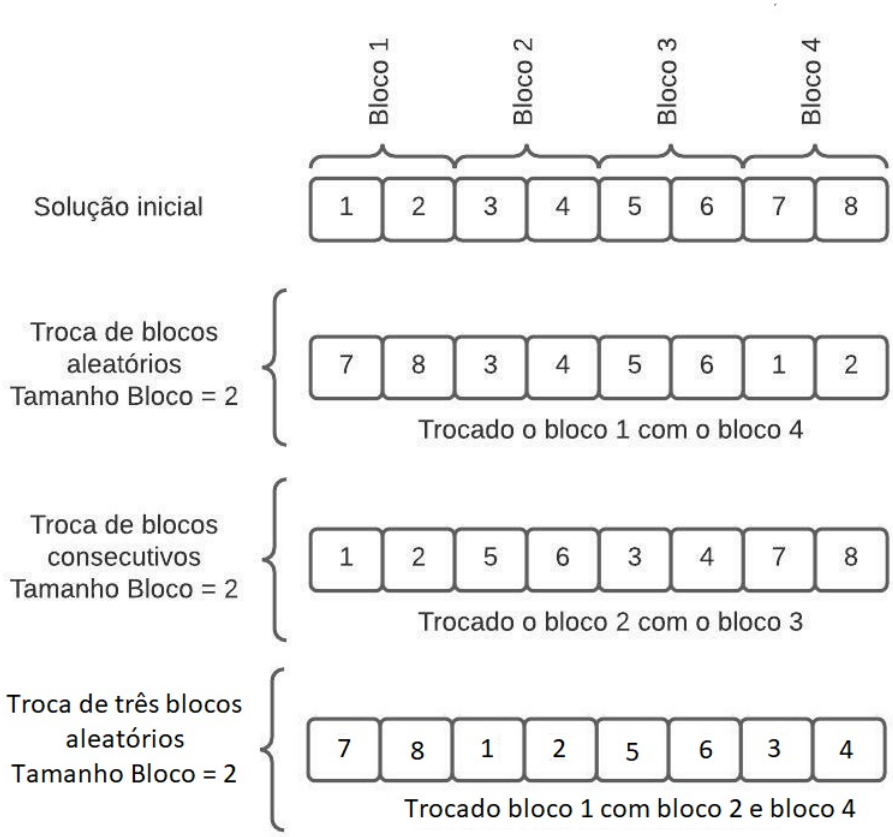
Quando uma solução já foi visitada e está salva na memória, o operador adaptativo realiza um dos movimentos apresentados para criar uma solução diferente da original e verifica-se se a solução criada já foi avaliada e está na estrutura de memória, caso esteja na estrutura de memória não é necessário calcular sua função objetivo, somente compara-se com a solução original, caso não esteja salva ela será avaliada e salva na memória. Pretende-se com este movimento contornar o problema do algoritmo ficar preso em uma determinada região do espaço de busca, o que permite uma melhor exploração de toda a região de soluções factíveis.

O operador proposto utiliza duas estruturas de vizinhança onde os movimentos são realizadas por blocos de tamanhos variáveis, sendo que o bloco inicia com tamanho dois e termina com a metade do tamanho da solução. A vizinhança utilizada será denotada por Na e realiza movimento de troca de dois blocos aleatórios. E a vizinhança que realiza movimento de

troca de dois blocos consecutivos será denotada por Nb.

A Figura 3.2 mostra um exemplo dos movimentos realizados com os blocos, ao invés da utilização de posições únicas para a realização dos movimentos é utilizado os blocos.

Figura 3.2 – Exemplo de funcionamento do operador adaptativo.



Conforme mostrado na Figura 3.2 o operador adaptativo recebe uma solução inicial, para este exemplo usamos o tamanho do bloco igual a 2. Para a troca aleatória de dois blocos, os dois blocos são escolhidos aleatoriamente, no exemplo foram selecionados os blocos 1 e 4, após essa troca temos uma outra solução diferente da inicial e isso faz com que exploremos outros lugares de busca.

Já para a troca de consecutivas de blocos, o primeiro bloco (bloco 1) é escolhido aleatoriamente e o segundo é o bloco 1 + 1 indicando que o bloco 2 é o bloco consecutivo ao bloco 1.Como mostrado na figura acima na troca consecutiva o valor para o bloco 1 foi 2 logo o bloco 2 é 3, sendo assim os blocos são movimentados gerando outra solução diferente da solução inicial e também diferente da solução achada na troca de blocos aleatórios explorando ainda mais a região do espaço de busca. Na troca de três blocos aleatórios, os três blocos são escolhidos aleatoriamente, no exemplo são escolhidos os probos 1, 2 e 4 e realizando essa troca temos uma solução diferente da solução inicial e isso nos leva a exploração outros lugares de busca.

3.3 VNS com Operador Adaptativo

O algoritmo apresenta o pseudocódigo do VNS com o operador adaptativo, nomeado como VNS-Oadp neste trabalho.

Algoritmo 3.3: VNS com Operador Adaptativo

Entrada: $N^{(.)}$, k_{max}, $f(.)$, s_i

início
- $s* \leftarrow s_i$
- inicializar memória M
- inserir s_i em M
- **faça**
  - $k \leftarrow 1$
  - **faça**
    - $s' \leftarrow t \in N^{(k)}(s*)$
    - **se** $s' \in M$ **então**
      - $f_{s'} \leftarrow M(s')$
      - $s^{\omega} \leftarrow buscaPorBloco(s')$
      - **se** $s^{\omega} \in M$ **então**
        - $f_{s^{\omega}} \leftarrow M(s^{\omega})$
      - **senão**
        - $f_{s^{\omega}} \leftarrow f(s^{\omega})$
        - inserir s^{ω} em M
      - **se** $f_s^{\omega} < f_s'$ **então**
        - $s' \leftarrow s^{\omega}$
    - **senão**
      - $f_{s'} \leftarrow f(s')$
      - inserir s' em M
    - $s \leftarrow buscaLocal(s')$
    - **se** $f(s) < f(s*)$ **então**
      - $s* \leftarrow s$
      - $k \leftarrow 1$
    - **senão**
      - $k \leftarrow k + 1$
  - **enquanto** $k \leq k_{max}$;
- **enquanto** *condição de parada não é satisfeita*;
- **retorna** s

Os parâmetros de entrada do algoritmo são a vizinhança ($N^{(.)}$), número máximo de

vizinhanças (k_{max}), a função objetivo ($f(.)$) e a solução inicial (s_i). O VNS parte da solução inicial $s*$ = s_i, inicializa a memória e insere s_i na memória.Enquanto não atingir o critério de parada, segue os seguintes passos: inicializa a estrutura de vizinhança $k = 1$, depois seleciona um vizinho aleatório s' de $s*$ pelo movimento k (linha 8); na linha 9 é verificado se s' está salvo na memória, se estiver o valor da sua função é resgatado da memória (linha 10) e na linha 11 é empregada uma busca por bloco na solução s' e esta busca é salva na variável s^{ω} (o operador adaptativo). Na linha 12 é verificado se s^{ω} está salva na memória, caso esteja o valor da função é buscada na memória, caso não esteja o valor da função é calculado e s^{ω} é salva na memória. Na linha 17 é feita a verificação se o valor da função de s^{ω} é menor que o valor de função em s', caso seja a solução inicial (s') é passa a ter o valor de s^{ω}. Na linha 19 é executada se a solução s' não estiver salva na memória, com isso é calculado o valor da função e inserida s' na memória. Já na linha 22 é realizada a busca local em cima da solução s'. Caso a solução resultante, s, for melhor que a solução atual $s*$, a solução $s*$ é atualizada (linhas 23 e 14). O processo executa uma nova busca local na solução atual, usando uma nova vizinhança e este processo é repetido até que o critério de parada seja satisfeito.

A busca local empregada para o VNS foi o VND, conforme descrito na Seção 2.1.2.1.

3.4 SA com operador Adaptativo

O algoritmo apresenta o pseudocódigo do SA com o operador adaptativo, nomeado como SA-Oadp neste trabalho..

Algoritmo 3.4: Simulated Annealing com Operador Adaptativo

Entrada: $N(.)$, $f(.)$, s, c, T
Saída: s

```
1  início
2      t_k ← T
3      inicializar memória M
4      inserir s em M
5      faça
6          faça
7              s' ← w ∈ N(s)
8              se s' ∈ M então
9                  f_s' ← M(s')
10                 s^ω ← buscaPorBloco(s')
11                 se s^ω ∈ M então
12                     f_s^ω ← M(s^ω)
13                 senão
14                     f_s^ω ← f(s^ω)
15                     inserir s^ω em M
16                 se f_s^ω < f_s' então
17                     s' ← s^ω
18             senão
19                 f_s' ← f(s')
20                 inserir s' em M
21             Δ_{s,s'} ← f(s') − f(s)
22             se Δ_{s,s'} ≤ 0 então
23                 s ← s'
24             senão
25                 s ← s' com probabilidade exp(−Δ_{s,s'}/t_k)
26         enquanto condição de parada não é satisfeita;
27         t_k ← t_k * (1 − c)
28     enquanto t_k ≥ 0;
29 retorna s
```

Os parâmetros de entradas são a vizinhança ($N(.)$), a função objetivo ($f(.)$), c é a taxa de resfriamento e T a temperatura inicial. Na linha 2 a variável t_k recebe a temperatura inicial passada

por parâmetro. Nas linhas 3 e 4 inicializa a memória e insere s_i na memória, respectivamente. Na linha 7 s' recebe um vizinho aleatório de s; já na linha 8 é verificado se a solução s' está na memória, caso esteja o valor da sua função é buscada na memória e é feita uma busca por blocos, a variável que recebe essa busca é chamada s^{ω}, caso a solução s^{ω} esteja salva na memória seu valor de função é buscado, caso não esteja seu valor de função é calculado e s^{ω} é inserida na memória; na linha 16 é comparado o valor de função de s^{ω} e de s', caso o da s^{ω} seja menor ele é atribuído para s'. Na linha 18 é executado quando s' não está na memória, seu valor de função é calculado e ela é salva na memória. Na linha 21 $\Delta_{s,s'}$ recebe a diferença do valor da solução nova com o valor da solução original e caso essa diferença seja menor ou igual a zero (linha 22) a solução nova é atribuída à solução original (linha 23). Caso contrário a solução nova é atribuída a solução original com uma probabilidade que é calculada por exponencial de $(-\Delta_{s,s'}/t_k)$ (linha 25). O processo da linha 3 até a linha 10 é repetido enquanto uma condição de parada não for satisfeita. Após a satisfação dessa condição a variável t_k recebe o seu valor multiplicado por 1 menos a taxa de resfriamento (linha 27). O processo da linha 5 até a linha 28 é repetido até que t_k seja menor ou igual a 0.

A busca local empregada para o SA foi o VND, conforme descrito na Seção 2.1.2.1.

4 Resultados

Neste capítulo são apresentados os resultados alcançados pelo método proposto neste trabalho. Os testes foram executados em um notebook Dell Inspirion-14 Série 3000, que possui processador Intel Core i5 de 1.6 GHz, memória RAM de 8GB. Foi utilizado o sistema operacional utilizado o Windows 10 e a linguagem Java versão 11, com o kit desenvolvimento OpenJDK 11. As implementações realizadas estão disponíveis no seguinte link <https://github.com/paulatoledo30/Monografia>.

O capítulo apresenta a seguinte estrutura: na Seção 4.1 são apresentados os problemas-testes usados para validar o método proposto. A Seção 4.2 exibe os resultados computacionais obtidos nas variantes do VNS e do SA e, por fim, na Seção 4.3 é apresentado a discussão dos resultados.

4.1 Problemas Teste

Para avaliar os métodos propostos serão utilizados dois problemas teste: o Problema de Agendamento de Tarefas com Data de Entrega Comum (Subseção 4.1.1) proposto por Biskup e Feldmann (2001) e uma variação do Problema do Caixeiro Viajante (Subseção 4.1.2) proposto por Carvalho et al. (2007). A escolha destes problemas de otimização combinatória se deu de forma aleatória e tem como objetivo medir a eficiência da metodologia proposta.

4.1.1 Problema de Agendamento de Tarefas com Data de Entrega Comum

Seja m uma máquina que executa uma única tarefa por vez, sem intervalos entre elas e seja n a quantidade de tarefas a serem executadas nessa máquina, todas com uma mesma data de entrega d. Cada tarefa i possui: (i) um tempo de processamento P_i, (ii) uma penalidade por adiantamento α_i e, (iii) uma penalidade por atraso β_i. O objetivo do problema é determinar a ordem em que as tarefas deverão ser executadas de modo a minimizar o somatório das penalidades.

Este problema possui duas versões: (i) uma não-restritiva, onde a data de entrega comum é uma variável de decisão e todas as tarefas podem ser entregues sem atraso e, (ii) uma restritiva, onde a data de entrega é externa, fornecida como parâmetro, e algumas tarefas poderão ser entregues com atraso. Existem alguns casos na versão não-restritiva que podem ser resolvidos de forma eficiente, mas para o caso geral, o problema considerado pertence a classe de problemas $\mathcal{NP}$-Difícil.

Para a formulação matemática do problema, considere os seguintes parâmetros:

α_i: penalidade por adiantamento da tarefa i, para $i = 1, \ldots, n$.

β_i: penalidade por atraso da tarefa i, para $i = 1, \ldots, n$.

p_i: tempo de processamento da tarefa i, para $i = 1, \ldots, n$.

d: data de entrega.

e as seguintes variáveis de decisão:

s_i: tempo de início da tarefa i, para $i = 1, \ldots, n$.

x_{ik}: variável de decisão que recebe 1 se a tarefa i é executada antes (imediatamente ou não) da tarefa k, para $i, k = 1, \ldots, n$; e 0 caso contrário.

Assim, o modelo matemático do problema, proposto por Biskup e Feldmann (2001), é dada por:

$$\text{minimizar:} \quad \sum_{i=1}^{n} \alpha_i E_i + \sum_{i=1}^{n} \beta_i T_i \tag{4.1}$$

$$\text{sujeito à:} \quad T_i \geq s_i + p_i - d \quad i = 1, \ldots, n \tag{4.2}$$

$$E_i \geq d - s_i - p_i \quad i = 1, \ldots, n \tag{4.3}$$

$$s_i + p_i \leq s_k + R(1 - x_{ik}), i = 1, \ldots, n; k = i + 1, \ldots, n. \tag{4.4}$$

$$s_k + p_k \leq s_i + Rx_{ik}, i = 1, \ldots, n - 1; k = i + 1, \ldots, n. \tag{4.5}$$

$$T_i, E_i, s_i \geq 0 \quad i = 1, \ldots, n \tag{4.6}$$

$$x_{ik} \in \{0, 1\} \quad i = 1, \ldots, n - 1; k = i + 1, \ldots, n \tag{4.7}$$

onde a Equação (4.1) representa a função objetivo que minimiza a soma dos custos das penalidades por antecipação ou por atraso da conclusão das tarefas. As restrições indicadas pela Equação (4.2) calculam o tempo de atraso de cada tarefa e as restrições indicadas pela Equação (4.3) calculam o tempo de antecipação de cada tarefa. As restrições representadas pelas Equações (4.4) e (4.5) determinam o tempo de início de cada tarefa, sendo R um número suficientemente grande. As Equações (4.6) e (4.7) definem o domínio das variáveis.

Para resolver o problema foi usado um conjunto de instâncias proposto por (BISKUP; FELDMANN, 2001). Este conjunto de instâncias é formado por sete subconjuntos de problemas, onde cada subconjunto possui 10 problemas diferentes em cada, gerando um total de 70 problemas testes. Cada conjunto é diferenciado por possuir uma quantidade específica de tarefas, que varia de 10, 20, 50, 100, 200, 500 e 1000 tarefas.

4.1.2 Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante consiste em estabelecer uma rota cíclica (começando e terminando no mesmo ponto) entre cidades, sem repetição, com o menor custo possível, a

ser percorrida por um vendedor viajante. Esse custo pode ser distância percorrida, quantidade de combustível gasto, dentre outros. Neste trabalho, foi utilizada uma variação do problema, conforme apresentada em (CARVALHO et al., 2007). Para tanto, considere $G = (V, A)$ um grafo completo não-direcionado e os seguintes parâmetros:

V: conjunto de vértices.

n: número de cidades, $n = |V|$.

c_{ij}: custo entre as cidades i e j; $i, j \in V$.

e a seguinte variável de decisão:

$$x_{ij} = \begin{cases} 1, & \text{se a cidade } i \text{ é visitado logo antes da cidade } j \\ 0 & \text{caso contrário;} \end{cases} \qquad \forall i, j \in V.$$

Neste contexto, o modelo matemático do problema é dado por:

$$\text{minimizar} \quad \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{4.8}$$

$$\text{sujeito à} \quad \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \ldots, n \tag{4.9}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1, \ldots, n \tag{4.10}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \quad \forall S \subset V \text{ e } S \neq \emptyset \tag{4.11}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V \tag{4.12}$$

em que a função objetivo, representada pela Equação (4.8), minimiza o somatório dos custos entre as cidades em toda a rota. As restrições representadas pelas Equações (4.9) e (4.10) garantem que toda cidade possua no máximo uma cidade anterior e uma cidade seguinte. As restrições representadas pela Equação (4.11) garantem a existência de uma solução factível e, por fim, as restrições representadas pela Equação (4.12) definem o domínio da variável de decisão.

O conjunto de instâncias utilizado do problema do Caixeiro Viajante é formado por cinco subconjuntos de problemas, com 5 problemas diferentes em cada um, gerando um total de 30 problemas. Cada conjunto é caracterizado por ter uma quantidade específica de tarefas que varia de 10, 20, 50, 100, 200 e 500. Por uma limitação de tempo para realizar todos os testes foram testados apenas 15 problemas de cada conjunto, sendo três problemas de tamanhos 20, 50, 100, 200 e 500.

4.2 Resultados Computacionais

Para realizar os testes computacionais, cada versão do algoritmo foi executada 10 vezes para cada instância do problema teste. Cada execução utilizou uma semente de números aleatórios distinta e a escolha dessa semente foi para garantir que todas as versões dos algoritmos partam do mesmo ponto em cada problema. Tal procedimento permite comparar o comportamento e a convergência de diferentes variações do algoritmo para um mesmo problema. As dez sementes, embora tenha aparência "binária", estão na base decimal e estão apresentadas na Tabela 4.1.

Tabela 4.1 – Sementes de números aleatórios utilizadas na execução do algoritmo.

Semente	Valor
1	1100111001
2	1100111010
3	1100111011
4	1100111100
5	1100111101

Semente	Valor
6	1100111110
7	1100111111
8	1101000000
9	1101000001
10	1101000010

As subseções seguintes apresentam os resultados para cada metaheurística/heurística utilizada e sua versão com o operador adaptativo. Os limitantes superiores de cada problema teste foram calculados utilizando as versões clássicas do VNS e do *Simulated Annealing*, além das versões clássicas foram utilizadas as versões dos algoritmos com a estratégia de memória proposta em (AMARO, 2019), que são:

VNS-MH: consiste na versão clássica do VNS com operador de memória.

SA-MH: consiste na versão clássica do SA com operador de memória.

Basicamente, o operador de memória proposto por Amaro (2019) se diferencia do proposto neste trabalho por armazenar as soluções e seu valor de função que já foram visitadas, impedindo assim um retrabalho do algoritmo de recalcular o valor de função toda vez que esbarra em uma solução igual, com isso diminuímos o tempo de excussão pois diversos problemas possuem funções objetivo muito trabalhosas e no presente trabalho são utilizadas as soluções armazenadas na memória para fazer movimentos por blocos nessas soluções gerando uma vizinhança diferente e diversa.

Neste contexto, os resultados obtidos pelo VNS e as suas variações (VNS-MH e VNS-Oadp) estão expostos na Subseção 4.2.1 e, os resultados alcançados pelo *Simulated Annealing* e suas variações (SA-MH e SA-Oadp) na Subseção 4.2.2.

4.2.1 Resultados para o VNS

Nesta seção são apresentados os resultados alcançados pelo VNS e as suas variações VNS-MH e VNS-Oadp para os problemas de Agendamento de Tarefas com Data de Entrega

Comum e o Caixeiro Viajante.

A Tabela 4.2 apresenta a solução média obtida por cada uma das variações do VNS nas 10 execuções realizadas no problema de Agendamento de Tarefas com Data de Entrega Comum. As colunas 2, 3 e 4, apresentam, respectivamente os resultados obtidos pela versão clássica do VNS, o VNS com estrutura de memória (VNS-MH), proposto por Amaro (2019), e o VNS-Oadp, proposto no presente trabalho.

Tabela 4.2 – Médias das soluções obtidas pelas versões do VNS para o problema de Agendamento de Tarefas com Data de Entrega Comum.

Problema-Teste	**VNS**	**VNS-MH**	**VNS-Oadp**
sch20k1	3452.4	**3244.4**	**3228.7**
sch20k2	5180.7	**5073.6**	**5010.3**
sch20k3	4267.1	**4003.2**	**4043.8**
sch50k1	25101.3	**24707.5**	**24410.6**
sch50k2	18824.5	**18474.5**	**18579.6**
sch50k3	21925.4	**21218.8**	**21502.6**
sch100k1	95364.6	**88258.4**	**87450.7**
sch100k2	76858.0	**74668.1**	**75421.1**
sch100k3	83266.8	**81671.3**	**81852.3**
sch200k1	317735.9	**303479.1**	**302878.2**
sch200k2	332254.1	**324841.0**	**327747.7**
sch200k3	304198.7	**299993.1**	**299761.6**
sch500k1	1945036.8	**1815824.2**	**1831178.0**
sch500k2	2147006.5	**2060563.4**	**2036152.4**
sch500k3	1975082.0	**1918370.0**	**1905613.2**

Na Tabela 4.2, temos em negrito as melhores soluções encontradas em relação ao VNS clássico e em negrito/azul temos as melhores soluções entre os três métodos.

Nesta tabela nota-se que o VNS-MH e o VNS-Oadp obtêm em todas as instâncias as melhores soluções em relação a versão clássica do VNS. Comparando o VNS-MH com o VNS-Oadp temos que o VNS-Oadp obtém oito melhores médias em relação ao VNS-MH e o VNS-MH é responsável por sete melhores soluções que o VNS-Oadp.

A Tabela 4.3 apresenta a média dos tempos obtidos durante as dez execuções do algoritmo para cada instância do problema de Agendamento de Tarefas com Data de Entrega Comum. Na tabela, a primeira coluna indica o problema-teste e nas demais colunas são indicados o tempo médio de execução dos algoritmos na seguinte ordem: VNS, VNS-MH e VNS-Oadp.

Tabela 4.3 – Tempo médio de execução, em segundos, das versões do VNS para o problema de Agendamento de Tarefas com Data de Entrega Comum.

Problema-Teste	**VNS**	**VNS-MH**	**VNS-Oadp**
sch20k1	0.0144	0.0380	0.0453
sch20k2	0.0153	0.0103	0.0118
sch20k3	0.0086	0.0181	0.0150
sch50k1	0.0620	0.1511	0.1658
sch50k2	0.0190	0.0982	0.0985
sch50k3	0.0209	0.1031	0.0863
sch100k1	0.1208	0.8968	1.1772
sch100k2	0.0950	0.8462	0.8039
sch100k3	0.1004	1.0093	0.9430
sch200k1	0.5199	8.0950	8.2229
sch200k2	0.5485	8.5982	8.5244
sch200k3	0.5200	9.0572	8.7592
sch500k1	3.9686	194.6172	160.3771
sch500k2	2.9210	141.3517	178.2848
sch500k3	3.2967	203.6622	173.8036

Em relação aos tempos médios apresentados, nota-se que o VNS-Oadp possui um tempo computacional maior em 7 das 15 instâncias e o VNS-MH nas outras 8 instâncias, mas vale lembrar que são os métodos que retornam os melhores resultados. O VNS clássico possui o menor tempo computacional, porém retorna os piores resultados.

As soluções médias obtida por cada uma das variações do VNS para as 10 execuções realizadas em cada problema do Caixeiro Viajante são apresentadas na Tabela 4.4. As colunas 2, 3 e 4, contém, respectivamente os resultados obtidos pela versão clássica do VNS, o VNS-MH e o VNS-Oadp. Na tabela é possível notar que o VNS-MH e o VNS-Oadp proposto neste trabalho obtém sempre soluções melhores que a versão clássica do VNS para o problema do Caixeiro Viajante.

Na tabela 4.4 temos em negrito as melhores soluções encontradas em relação ao VNS clássico e em negrito/azul temos as melhores soluções entre os três métodos. Nesta tabela nota-se que o VNS-MH e o VNS-Oadp obtêm em todas as instâncias as melhores soluções em relação a versão clássica do VNS. Comparando o VNS-MH com o VNS-Oadp temos que o VNS-Oadp obtém dez melhores médias em relação ao VNS-MH e o VNS-MH é responsável por cinco melhores soluções que o VNS-Oadp.

Tabela 4.4 – Médias das soluções obtidas pelas versões do VNS para o problema do Caixeiro Viajante.

Problema-Teste	VNS	VNS-MH	VNS-Oadp
cv20k1	216639.8	**177321.8**	**200779.1**
cv20k2	193713.9	**175943.2**	**166470.2**
cv20k3	201227.0	**166692.7**	**167432.1**
cv50k1	473954.6	**459821.1**	**461985.5**
cv50k2	498126.2	**444307.5**	**447590.0**
cv50k3	498249.7	**422654.8**	**408963.8**
cv100k1	1029521.6	**947359.7**	**927719.2**
cv100k2	1089624.9	**909451.3**	**883301.7**
cv100k3	984931.4	**937000.0**	**926189.0**
cv200k1	2244722.5	**1857210.0**	**1971466.8**
cv200k2	2215101.2	**2046461.6**	**1951957.8**
cv200k3	2242053.5	**2009644.4**	**1922341.2**
cv500k1	5689366.0	**5071996.0**	**4900363.0**
cv500k2	5689366.0	**5071996.0**	**4900363.0**
cv500k3	5689366.0	**5071996.0**	**4900363.0**

A Tabela 4.5 apresenta a média dos tempos obtidos durante as dez execuções do algoritmo para cada instância do Problema do Caixeiro Viajante. Na tabela, a primeira coluna indica o problema teste e nas demais colunas são indicados o tempo médio de execução dos algoritmos na seguinte ordem: VNS, VNS-MH e VNS-Oadp.

Tabela 4.5 – Média dos tempos de execução, em segundos, das 10 execuções de cada método para o problema do Caixeiro Viajante.

Problema-Teste	VNS	VNS-MH	VNS-Oadp
cv20k1	0.0057	0.0256	0.0081
cv20k2	0.0063	0.0255	0.0170
cv20k3	0.0045	0.0125	0.0154
cv50k1	0.0047	0.0394	0.0629
cv50k2	0.0056	0.0295	0.0160
cv50k3	0.0016	0.0149	0.0254
cv100k1	0.0136	0.1178	0.2297
cv100k2	0.0015	0.1026	0.1491
cv100k3	0.0048	0.1024	0.0899
cv200k1	0.0188	0.6395	0.5198
cv200k2	0.0147	0.4450	0.4984
cv200k3	0.0071	0.4516	0.4860
cv500k1	0.0999	9.2899	6.1373
cv500k2	0.0703	9.2686	6.1394
cv500k3	0.0711	7.2655	5.8334

Em relação aos tempos médios nota-se que o VNS-Oadp possui um tempo computacional maior em 7 das 15 instâncias e o VNS-MH nas outras 8 instâncias, mas vale lembrar que

são os métodos que retornam os melhores resultados. O VNS clássico possui o menor tempo computacional, porém retorna os piores resultados.

4.2.2 Resultados para o *Simulated Annealing*

Nesta seção são apresentados os resultados alcançados pelo *Simulated Annealing* e as suas variações SA-MH e SA-Oadp para os problemas de Agendamento de Tarefas com Data de Entrega Comum e Caixeiro Viajante.

A Tabela 4.6 apresenta a solução média obtida por cada uma das variações do SA nas 10 execuções realizadas no problema de Agendamento de Tarefas com Data de Entrega Comum. As colunas 2, 3 e 4, apresentam, respectivamente os resultados obtidos pela versão clássica do SA, o SA com estrutura de memória (SA-MH) proposto por Amaro (2019) e o SA-Oadp proposto no presente trabalho.

Tabela 4.6 – Médias das soluções obtidas pelas versões do SA para o problema de Agendamento de Tarefas com Data de Entrega Comum.

Problema-Teste	**SA**	**SA-MH**	**SA-Oadp**
sch20k1	4508.3	**4441.2**	**4411.3**
sch20k2	6182.1	6282.4	**6172.8**
sch20k3	5604.9	**5596.5**	**5561.4**
sch50k1	40506.9	**40334.3**	40390.6
sch50k2	34503.5	**34211.6**	34853.0
sch50k3	38388.0	38453.8	**38040.3**
sch100k1	176358.3	**175504.4**	176204.7
sch100k2	155765.6	**154900.4**	**153938.1**
sch100k3	161481.6	**160658.9**	**159972.6**
sch200k1	691912.6	**685175.4**	691458.2
sch200k2	731078.9	731078.9	731078.9
sch200k3	673878.5	675936.4	**670862.6**
sch500k1	4435019.0	4435019.0	4435019.0
sch500k2	4821072.0	4821072.0	4821072.0
sch500k3	4629069.0	4629069.0	4629069.0

Na Tabela 4.6 temos em negrito as melhores soluções encontradas em relação ao SA clássico e em negrito/azul temos as melhores soluções entre os três métodos.

Como apresentado na Tabela 4.6, quando comparados com o *Simulated Annealing*, o SA-Oadp obteve um resultado melhor em 7 instâncias e o SA-MH obteve os melhores resultados em 4 instâncias. Para 4 instâncias os resultados obtidos por todas as três versões do SA foram iguais. Os piores resultados foram obtidos pela versão clássica do SA.

A Tabela 4.7 apresenta a média dos tempos obtidos durante as dez execuções do algoritmo para cada instância do problema de Agendamento de Tarefas com Data de Entrega Comum. Na tabela, a primeira coluna indica o problema-teste e nas demais colunas são indicados o tempo

médio de execução dos algoritmos SA, SA-MH e SA-Oadp. Nota-se que, em relação ao tempo computacional, o SA-MH e SA-Oadp na maioria das vezes encontrou as soluções com um custo computacional bem maior que o SA clássico, mas como dito anteriormente as variações obtêm as melhores soluções.

Tabela 4.7 – Média dos tempos de execução, em segundos, das 10 execuções de cada método para o problema de Agendamento de Tarefas com Data de Entrega Comum

Problema-Teste	**SA**	**SA-MH**	**SA-Oadp**
sch20k1	0.1378	0.1171	0.0665
sch20k2	0.0513	0.0428	0.0344
sch20k3	0.0328	0.0470	0.0336
sch50k1	0.1396	0.1696	0.1988
sch50k2	0.1180	0.1133	0.1355
sch50k3	0.0725	0.0998	0.1384
sch100k1	0.1518	0.3714	0.4787
sch100k2	0.1089	0.2835	0.3951
sch100k3	0.1016	0.2635	0.3599
sch200k1	0.2795	1.2279	1.7869
sch200k2	0.2071	1.1171	1.5915
sch200k3	0.2046	1.0293	1.4777
sch500k1	0.6060	6.6807	6.3834
sch500k2	0.5738	6.4216	6.6143
sch500k3	0.5513	7.4524	6.3599

Na Tabela 4.8 temos em negrito as melhores soluções encontradas em relação ao SA clássico e em negrito/azul temos as melhores soluções entre os três métodos.

A Tabela 4.8 apresenta a solução média obtida por cada uma das variações do SA nas 10 execuções realizadas no problema do Caixeiro Viajante. As colunas 2, 3 e 4, apresentam, respectivamente os resultados obtidos pela versão clássica do SA, o SA com estrutura de memória (SA-MH) proposto por Amaro (2019) e o SA-Oadp.

Como apresentado na Tabela 4.8, nota-se que o SA-Oadp obteve um resultado melhor em 7 instâncias, o SA-MH obteve os melhores resultados em 2 instâncias, em 3 instâncias o resultado foi o mesmo nas três versões do SA e a versão clássica do SA obteve 3 melhores resultados em relação aos outros dois métodos.

Tabela 4.8 – Médias das soluções obtidas pelas versões do SA para o problema do Caixeiro Viajante.

Problema-Teste	SA	SA-MH	SA-Oadp
cv20k1	**189307.6**	193258.9	194747.0
cv20k2	182054.2	184546.8	**175253.7**
cv20k3	177054.2	181980.3	**174695.4**
cv50k1	588073.9	**561392.6**	**579204.6**
cv50k2	568268.1	566397.9	**555611.7**
cv50k3	**552930.2**	561544.4	567841.5
cv100k1	1306121.5	**1304026.0**	**1300655.2**
cv100k2	1293113.9	1298263.1	**1293759.5**
cv100k3	**1327560.4**	1331166.2	1334584.6
cv200k1	2858808.8	**2858207.5**	2860018.8
cv200k2	2847723.8	2848865.0	**2846621.5**
cv200k3	2852582.5	**2846869.2**	**2845688.0**
cv500k1	7632800.0	7632800.0	7632800.0
cv500k2	7632800.0	7632800.0	7632800.0
cv500k3	7632800.0	7632800.0	7632800.0

A Tabela 4.9 apresenta a média dos tempos obtidos durante as dez execuções do algoritmo para cada instância do problema do Caixeiro Viajante. Na tabela, a primeira coluna indica o problema-teste e nas demais colunas são indicados o tempo médio de execução dos algoritmos SA, SA-MH e SA-Oadp.

Tabela 4.9 – Média dos tempos de execução, em segundos, das 10 execuções de cada método para o problema do Caixeiro Viajante.

Problema-Teste	SA	SA-MH	SA-Oadp
cv20k1	0.0158	0.0395	0.1160
cv20k2	0.0079	0.0222	0.0661
cv20k3	0.0082	0.0210	0.0537
cv50k1	0.0248	0.0512	0.0904
cv50k2	0.0145	0.0628	0.0655
cv50k3	0.0133	0.0858	0.0642
cv100k1	0.0703	0.2596	0.2449
cv100k2	0.0721	0.2217	0.2280
cv100k3	0.0361	0.2034	0.2364
cv200k1	0.0804	0.9735	1.1275
cv200k2	0.0520	0.9125	1.0324
cv200k3	0.0386	0.9544	0.9710
cv500k1	0.3828	6.7131	7.2431
cv500k2	0.4367	6.7798	5.7945
cv500k3	0.3268	6.4824	5.8143

Em relação ao custo computacional, o SA-Oadp tem o maior tempo em 11 das 15 instâncias e SA-MH tem o maior tempo em 4 das 15 instâncias. Já o SA clássico apresenta o

melhor tempo em relação as variações SA-MH e SA-Oadp, mas é importante ressaltar que o SA obteve soluções piores que suas variações.

4.3 Discussão dos Resultados

Neste trabalho, foi proposto uma atualização do operador adaptativo de busca local proposto por (COELHO, 2016) e realizado um estudo do impacto deste operador quando aplicado na heurística *Simulated Annealing* e na metaheurística VNS. Para testar a eficiência do operador, os resultados obtidos para os problemas testes, descritos na Seção 4.1, foram comparados com os resultados obtidos pelo limitante superior e pelas versões dos mesmos algoritmos propostas em (AMARO, 2019).

O VNS-Oadp melhorou quinze casos em relação ao VNS clássico e melhorou em oito casos em relação ao VNS-MH, isso pode ter ocorrido pelo fato de que o operador adaptativo implementado divide a da solução em blocos de tamanho variáveis e utiliza três estruturas de vizinhança.

Em relação ao tempo computacional a diferença entre os três métodos é pequeno, o que não interfere muito na escolha de qual método usar deixando somente o número de soluções melhores como parâmetro para essa escolha.

O tempo na heurística SA e suas versões é bem melhor que o tempo na metaheurística VNS e suas versões para o problema de Agendamento de Tarefas com Data de Entrega em Comum, porém o VNS em suas três versões apresenta médias de soluções muito melhores que o SA em suas três versões.

No problema problema do Caixeiro Viajante utilizando a metaheurística VNS, podemos analisar que em 10 das 15 instâncias a versão do VNS-Oadp apresentou melhores resultados médios em relação aos outros dois métodos. No VNS-MH tivemos 5 das 15 instâncias um resultado melhor do que o VNS clássico e o VNS-Oadp. Em relação ao tempo computacional VNS clássico apresenta um tempo menor em relação aos outros dois métodos, mas vale ressaltar que tanto o VNS-MH e o VNS-Oadp obtêm as melhores médias de solução mesmo o tempo sendo um pouco maior.

Em relação ao SA para o problema do Caixeiro Viajante, podemos observar que em 7 das 15 instâncias o método SA-Oadp apresenta melhor média de soluções em relação aos outros dois métodos. O SA-MH apresenta 2 soluções melhores e 3 soluções iguais em relação ao SA clássico e também ao SA-Oadp. O SA clássico para esse problema apresenta 3 soluções melhores que os outros dois métodos.

No que se refere ao tempo computacional temos que o SA-MH e o SA-Oadp apresentam um tempo maior de execução, porém apresentam as melhores médias de solução. Podemos perceber que para instâncias pequenas o tempo fica bem próximo nos três métodos apresentados.

O tempo na heurística SA e suas versões é bem parecido com o tempo na metaheurística VNS e suas versões para o problema do Caixeiro Viajante, porém o VNS em suas três versões apresenta médias de soluções muito melhores que o SA em suas três versões. E utilizando o SA na instância de tamanho 500 temos uma solução igual para os três métodos e no VNS não é apresentado nenhuma solução igual nos três métodos.

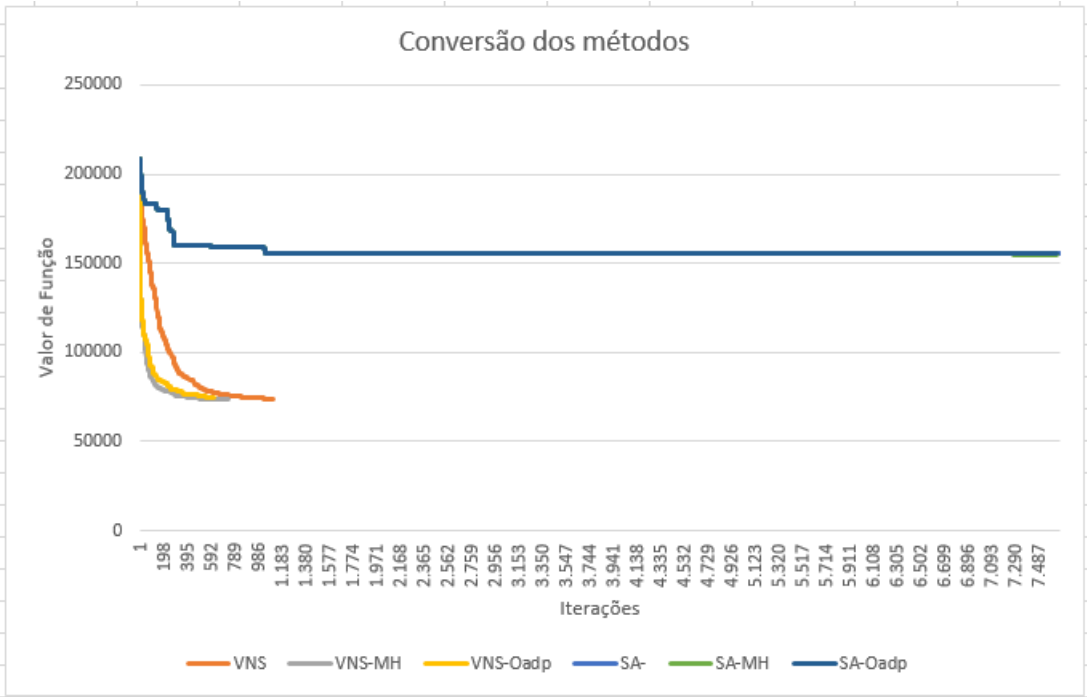
Os resultados indicam que a versão do VNS-Oapd apresentou resultado melhor em oito instâncias comparados a sete melhores resultados do VNS-MH para o problema de Agendamento de Tarefas com Data de Entrega em Comum e para o problema do Caixeiro Viajante apresentou resultados melhores em dez instâncias comparados a cinco melhores resultados do VNS-MH, porém a quantidade de testes realizados ainda não é suficiente para se chegar a tal conclusão. Mas podemos concluir que tanto o VNS-MH quanto o VNS-Oapd obtém melhores resultados que o VNS clássico. Podemos perceber também que o problema influencia na quantidade de soluções melhores obtidas pelo VNS-Oadp em relação ao VNS-MH.

Já o SA nos dois problemas-teste podemos perceber que o SA-Oadp obteve resultados melhores em sete instâncias, porém no problema de Agendamento de Tarefas com Data de Entrega em Comum o SA clássico não teve nenhum melhor resultado e o SA-MH obteve quatro melhores resultados e no problema do Caixeiro Viajante o SA clássico obteve três melhores soluções em relação aos outros dois métodos e o SA-MH obteve somente em duas instâncias os melhores resultados. Nas instâncias de tamanho 500 nos dois problemas teste o resultado dos três métodos foi o mesmo, porém para o problema de Agendamento a instância 200k2 também obteve a mesma média de resultados nos três métodos.

Outro ponto relevante a se considerar é a curva de convergência dos algoritmos. A Figura 4.1 apresenta a convergência das três variações do VNS e do SA para o problema de Agendamento de Tarefas com Data de Entrega Comum, considerando a instância sch100k2. Pode-se observar na figura que qualquer variação do VNS converge mais rápido e para uma solução melhor do que qualquer variação do SA. Em relação as variações do VNS-MH e VNS-Oadp, estas convergem mais rápido que o VNS clássico. O VNS-Oadp tem uma convergência menor, porém próxima, do VNS-MH.

Em relação ao SA e sua variações, nota-se na figura que a curva de convergência foi bem parecida para as três versões. Quase não é possível notar diferença entre os métodos. Para todas as variações do SA o número de iterações foi muito alta e o valor de função pior, quando comparados com o VNS. O que pode indicar que o efeito do operador de memória e do operador adaptativo no VNS foi mais eficaz que no SA.

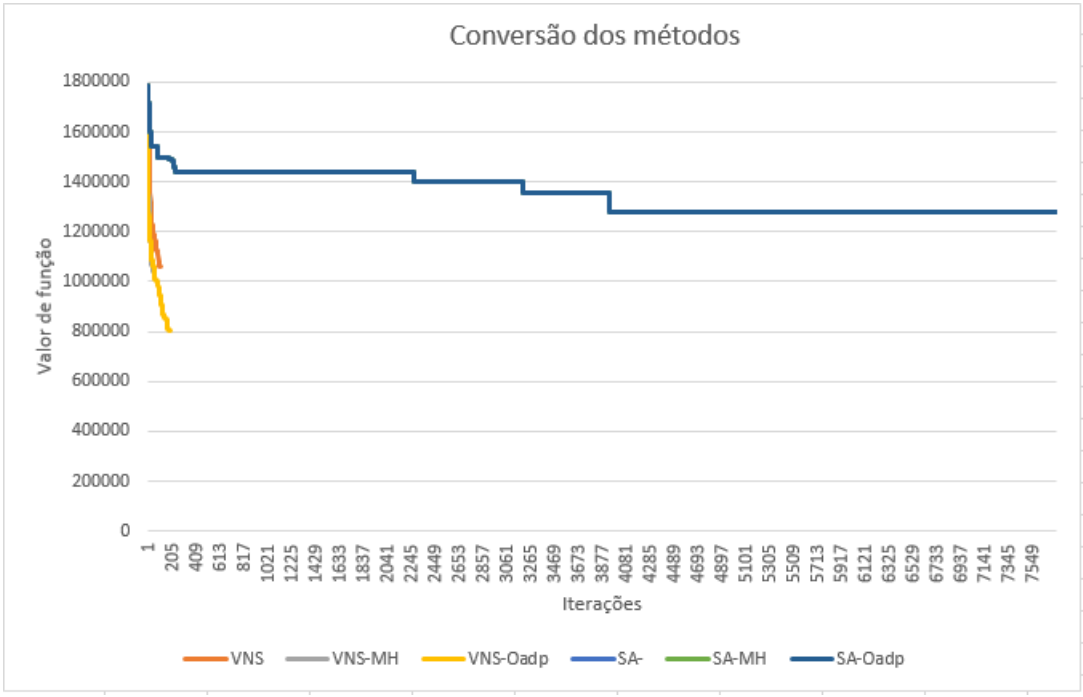
Figura 4.1 – Convergência dos algoritmos considerando a instância sch100k2 do problema problema de Agendamento de Tarefas com Data de Entrega em Comum



Na Figura 4.2 mostra a curva de convergência das três variações do VNS e do SA para o problema do Caixeiro Viajante, considerando na instância cv100k2. Na figura nota-se que qualquer variação do VNS converge mais rápido e para uma solução melhor do que qualquer variação do SA. O VNS-Oadp, para este problema de exemplo, obteve uma resposta melhor com poucas iterações quando comparados as outras versões do VNS.

Em relação ao SA e sua variações nota-se que todas as variações obtiveram a mesma curva de convergência, o que pode indicar que os operadores de memória e adaptativa não geraram o efeito esperado para esta heurística. Quando comparados com as variações do VNS, as versões do SA se mostraram inferiores tanto em relação ao valor de função encontrado quanto em relação a convergência dos algoritmos.

Figura 4.2 – Convergência dos algoritmos considerando o problema teste cv100k2 do problema do Caixeiro Viajante.



5 Considerações Finais

Neste trabalho, foi proposto uma melhoria ao operador adaptativo proposto por Coelho (2016) e estudado o efeito deste operador quando aplicado a metaheurística VNS e a heurística *Simulated Annealing*. As diferenças entre os operadores deste trabalho e da literatura se encontram no fato de que: (i) no operador adaptativo implementado neste trabalho, o tamanho do bloco cresce, de um em um, ao longo da adaptação enquanto na metodologia proposta por Coelho (2016) o tamanho deste bloco decresce, de um em um; (ii) no operador proposto, a estrutura de memória utilizada foi desenvolvida por Amaro (2019); e (iii) Coelho (2016) aplica o operador apenas na metaheurística VNS e valida em apenas um problema teste.

As versões propostas neste trabalho, VNS-Oapd e SA-Oapd foram comparadas com as versões clássicas do VNS e SA e com as versões propostas por Amaro (2019), VNS-MH e SA-MH. Testes computacionais utilizando dois problemas clássicos da literatura foram realizados a fim de verificar o desempenho da metodologia proposta.

Para o VNS, os testes mostraram que o VNS-MH e o VNS-Oapd obtêm melhores resultados quando comparados com o VNS clássico. O VNS-Oapd proposto se mostrou bem parecido com o VNS-MH, visto que o VNS-Oadp obteve soluções melhores para a oito instâncias enquanto o VNS-MH apresentou soluções melhores para sete instâncias do problema de Agendamento de Tarefas com Data de Entrega em Comum. Para o problema do Caixeiro Viajante o VNS-Oadp proposto mostrou-se melhor que o VNS-MH, uma vez que obteve resultados melhores para dez dos problemas enquanto o VNS-MH e o VNS-MH para apenas cinco problemas. No geral, o operador proposto, melhorou a qualidade das soluções obtidas quando comparados a versão clássica do VNS com uma convergência, mais rápida.

Em relação as variações do *Simulated Annealing*, a melhor versão indicada pelos testes para o problema Agendamento de Tarefas com Data de Entrega em Comum foi a SA-Oadp que apresentou sete soluções melhores, quatro soluções piores e quatro soluções iguais, quando comparados ao SA-MH e se mostrou melhor que o SA clássico o SA-Oadp em dez problemas. Para o problema do Caixeiro Viajante, a melhor versão indicada pelos testes foi o SA-Oadp que obteve sete soluções melhores que os outros dois métodos. Para o problema do Caixeiro Viajante o SA clássico foi melhor que o SA-MH e o SA-Oadp em três instâncias.

Quando comparadas as versões do *Simulated Annealing* com o VNS, nota-se que todas as versões do *Simulated Annealing* apresentaram valores de médias inferiores que o VNS e também uma convergência mais lenta.

Neste contexto, observa-se que o operador de busca local proposto se mostrou eficiente quando aplicado ao VNS por obter melhores soluções com convergência mais rápida que a versão clássica da metaheurística. Porém a versão proposta não se mostrou tão melhor que a versão

apenas com memória desenvolvida por Amaro (2019). Tal fato, nos permite concluir que ainda é preciso melhorar o operador para utilizar as informações armazenadas na memória de forme que se faça uma exploração melhor do espaço de busca e aumente o desempenho do algoritmo. Para a heurística *Simulated Annealing* as variações propostas não se mostraram eficientes, o que nos permite concluir que para ser inserido em diferentes algoritmos, o operador adaptativo precisa ser melhor estudado.

Apesar dos resultados ainda não serem totalmente satisfatórios, é possível notar que a estratégia de memória e o operador de busca local adaptativo podem ser ótimas estratégias para melhorar o desempenho de heurísticas e metaheurísticas clássicas, fazendo com que o estudo nessa área seja vantajoso. Como perspectivas para trabalhos futuros, consideramos:

- implementação de mais estruturas de vizinhança no operador adaptativo de busca local;
- implementação de outra estrutura de memória que permita fazer uma análise melhor do espaço de soluções;
- adaptar o operador para diferentes heurísticas e/ou metaheurísticas;
- importação do código fonte da aplicação para linguagens focadas em alto desempenho;
- testes computacionais utilizando diferentes problemas testes;
- aplicação da metodologia proposta em outras heurísticas e metaheurísticas.

Referências

AMARO, M. J. J. *Estudo Comparativo da Aplicação de Estruturas de Memória em Metaheurísticas de Busca Local*. Monografia (Iniciação Científica) — Universidade Federal de Ouro Preto, Ouro Preto, ago. 2019.

BISKUP, D.; FELDMANN, M. Benchmarks for scheduling on a single machine against restrictive and unrestrictive common due dates. *Computers & Operations Research*, Elsevier, v. 28, n. 8, p. 787–801, 2001.

CARRANO, E. G.; MOREIRA, L. A.; TAKAHASHI, R. H. A new memory based variable-length encoding genetic algorithm for multiobjective optimization. In: SPRINGER. *International Conference on Evolutionary Multi-Criterion Optimization*. [S.l.], 2011. p. 328–342.

CARVALHO, M. B. de et al. *Aplicações de meta-heuristica genetica e fuzzy no sistema de colonia de formigas para o problema do caixeiro viajante*. Dissertação (Mestrado) — Universidade Estadual de Campinas, 2007.

CHOW, C. K.; YUEN, S. Y. A non-revisiting particle swarm optimization. In: IEEE. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. [S.l.], 2008. p. 1879–1885.

CHOW, C. K.; YUEN, S. Y. An evolutionary algorithm that makes decision based on the entire previous search history. *IEEE Transactions on Evolutionary Computation*, IEEE, v. 15, n. 6, p. 741–769, 2011.

COELHO, D. G. *Um Algoritmo Evolutivo Multiobjetivo Aplicado ao Problema de Corte Bidimensional Guilhotinado*. Dissertação (mathesis) — Centro Federal de Educação Tecnológica de Minas Gerais, Belo Horizonte, 2011.

COELHO, D. G. *Estruturas de Memória e Operadores Adaptativos em Meta-Heurísticas*. Tese (Doutorado) — Universidade Federal de Minas Gerais, 2016.

FIALHO Álvaro R. S. *Adaptive operator selection for optimization*. Tese (Doutorado) — Université Paris-Sud 11, dez. 2010.

FREIRE, P. Pedagogia do oprimido. *Ed. Rio de Janeiro: Paz e Terra*, v. 3, p. 155, 1987.

FUCHS, H.; KEDEM, Z. M.; NAYLOR, B. F. On visible surface generation by a priori tree structures. In: ACM. *ACM Siggraph Computer Graphics*. [S.l.], 1980. v. 14, n. 3, p. 124–133.

GERALDO, D.; MARCHI, M. M. de. Aprimoramento da metaheurÍstica multiobjective harmony search para problemas com variÁveis continuas. 2012.

GILOVICH, T.; GRIFFIN, D.; KAHNEMAN, D. *Heuristics and biases: The psychology of intuitive judgment*. [S.l.]: Cambridge university press, 2002.

GLOVER, F.; KOCHENBERGER, G. *Handbook of Metaheuristics*. [S.l.]: Kluwer Academics Publisher, 2003. ISBN 1-4020-7263-5.

GONçALVES, R. A. et al. Seleção adaptativa de operadores aplicada ao problema do despacho econômico de energia elétrica. SEMISH, 2016.

GONçALVES, R. A. et al. Adaptive operator selection for many-objective optimization with nsga-iii. Conferência Internacional sobre Otimização Evolutiva Multi-Critério, 2017.

GUNAWAN, A.; LAU, H. C.; LU, K. Adopt: Combining parameter tuning and adaptive operator ordering for solving a class of orienteering problems. Computers Industrial Engineering, 2018.

JIANG, Y. et al. An artificial bee colony with self-adaptive operators and alterable search depth approach for intercell scheduling. Congress on Evolutionary Computation, 2016.

KRAMER, O. Evolutionary self-adaptation: a survey of operators and strategy parameters. *Evolutionary Intelligence*, Springer, v. 3, n. 2, p. 51–65, 2010.

NAVAS, M. M.; URBANEJA, A. J. N. Metaheuristicas multiobjetivo adaptativas. *Computación y Sistemas*, Centro de Investigacion en Computacion, IPN, v. 17, n. 1, p. 53–62, 2013.

PONGCHAIRERKS, P. An enhanced two-level metaheuristic algorithm with adaptive hybrid neighborhood structures for the job-shop scheduling problem. *Complexity*, Hindawi, 2020.

RIBEIRO, F. F.; SOUZA, S. R. de; SOUZA, M. J. F. ResoluÇÃo do problema de sequenciamento em uma mÁquina com minimizaÇÃo das penalidades por antecipaÇÃo e atraso da produÇÃo atravÉs de algoritmo genÉtico adaptativo. 2010.

SOUZA, M. J. F. *Inteligência Computacional para Otimização : Notas de aula*. [S.l.], 2009. Disponível em: <http://www.iceb.ufop.br/decom/prof/marcone>.

ŠRAMKO, A. *Enhancing a genetic algorithm by a complete solution archive based on a trie data structure*. [S.l.]: na, 2009.

THIERENS, D. Adaptive mutation rate control schemes in genetic algorithms. In: IEEE. *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*. [S.l.], 2002. v. 1, p. 980–985.

YUEN, S. Y.; CHOW, C. K. A non-revisiting genetic algorithm. In: *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. [S.l.: s.n.], 2007. p. 4583 –4590.

YUEN, S. Y.; CHOW, C.-K. A non-revisiting genetic algorithm. *IEEE Transactions on Evolutionary Computation*, p. 4583–4590, 2007.

YUEN, S. Y.; CHOW, C. K. Applying non-revisiting genetic algorithm to traveling salesman problem. In: IEEE. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. [S.l.], 2008. p. 2217–2224.

YUEN, S. Y.; CHOW, C.-K. A non-revisiting simulated annealing algorithm. In: IEEE. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. [S.l.], 2008. p. 1886–1892.

YUEN, S. Y.; CHOW, C. K. Continuous non-revisiting genetic algorithm. In: IEEE. *2009 IEEE Congress on Evolutionary Computation*. [S.l.], 2009. p. 1896–1903.

YUEN, S. Y.; CHOW, C. K. A genetic algorithm that adaptively mutates and never revisits. *IEEE transactions on evolutionary computation*, IEEE, v. 13, n. 2, p. 454–472, 2009.

ZIVIANI, N. Projeto de algoritmos com implementação e c++ e java. *THOMPSON Learning, São Paulo, 1st edição*, 2007.