Trabalho de Conclusão de Curso

Uma matheurística Construct, Merge, Solve & Adapt para o Problema da Cadeia de Caracteres Mais Próxima

Emily Brito de Oliveira
ebo@ic.ufal.br

Orientador:
Rian Gabriel Santos Pinheiro

Maceió, Abril de 2024

Emily Brito de Oliveira

Uma matheurística Construct, Merge, Solve & Adapt para o Problema da Cadeia de Caracteres Mais Próxima

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Rian Gabriel Santos Pinheiro

Maceió, Abril de 2024

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Helena Cristina Pimentel do Vale – CRB4 - 661

O48u Oliveira, Emily Brito de.
Uma matheurística Construct, Merge, Solve & Adapt para o problema da cadeia de caracteres mais próxima / Emily Brito de Oliveira. – 2024.
51 f.: il.

Orientador: Rian Gabriel Santos Pinheiro.
Monografia (Trabalho de Conclusão de Curso em Ciência da Computação) – Universidade Federal de Alagoas, Instituto de Computação. Graduação em Ciência da Computação. Maceió, 2024.

Bibliografia: f. 44-51.

1. Meta-heurísticas. 2. Problema da cadeia de caracteres mais próxima. 3. Construct, Merge, Solve & Adapt. 4. Busca tabu. 5. Auto-adaptação. I. Título.

CDU: 004.421

,

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Instituto de Computação – IC
Campus A. C. Simões - Av. Lourival de Melo Mota, BL 12
Tabuleiro do Martins, Maceió/AL - CEP: 57.072-970
Telefone: (082) 3214-1401

Trabalho de Conclusão de Curso - TCC

Formulário de Avaliação

Nome do Aluno

E	M	I	L	Y		B	R	I	T	O		D	E		O	L	I	V	E
I	R	A																	

Nº de Matrícula

					2	0	1	1	0	8	9	-	7

Título do TCC (Tema)

Uma matheurística Construct, Merge, Solve & Adapt para o Problema da Cadeia de Caracteres Mais Próxima

Banca Examinadora

Rian Gabriel Santos Pinheiro
Nome do Orientador

Documento assinado digitalmente
gov.br RIAN GABRIEL SANTOS PINHEIRO
Data: 15/04/2024 11:57:30-0300
Verifique em https://validar.iti.gov.br

Bruno Costa e Silva Nogueira
Nome do Professor

Documento assinado digitalmente
gov.br BRUNO COSTA E SILVA NOGUEIRA
Data: 15/04/2024 17:24:40-0300
Verifique em https://validar.iti.gov.br

Bruno José da Silva Barros
Nome do Professor

Documento assinado digitalmente
gov.br BRUNO JOSE DA SILVA BARROS
Data: 16/04/2024 02:45:27-0300
Verifique em https://validar.iti.gov.br

Assinatura

Data da Defesa: 15/04/2024

Nota Obtida: 10 (Dez)

Observações

Coordenador do Curso
De Acordo

Documento assinado digitalmente
gov.br ROBERTA VILHENA VIEIRA LOPES
Data: 29/04/2024 17:45:17-0300
Verifique em https://validar.iti.gov.br

Assinatura

Resumo

O Problema da Cadeia de Caracteres Mais Próxima (PCCP) é um problema de otimização combinatória que busca determinar uma *string* que mais se aproxima de um conjunto de *strings* de mesma dimensão. O PCCP faz parte de um conjunto de diversos problemas de otimização que envolvem comparação de *strings* e que encontram aplicações na Biologia Molecular. Este trabalho propõe um algoritmo *Self-Adaptive Construct, Merge, Solve & Adapt* (Adapt-CMSA) utilizando mecanismos da meta-heurística *Tabu Search* para resolver o PCCP. O CMSA é uma *matheurística* que explora características do problema para gerar uma sub-instância reduzida a partir da instância original, aplicando, em seguida, um algoritmo exato para resolvê-la. A versão *self-adaptive* do CMSA incrementa o algoritmo com mecanismos de auto-adaptação de seus parâmetros, tornando-o menos sensível à sua configuração em diferentes instâncias. Resultados experimentais em instâncias artificiais e reais demonstram que a abordagem proposta apresenta um desempenho significativamente superior em relação aos métodos heurísticos recentes propostos na literatura. O valor ótimo é encontrado na maioria das instâncias e o *gap* médio é consideravelmente reduzido. Além disso, são realizadas análises estatísticas com objetivo de avaliar a importância dos componentes do algoritmo no seu bom desempenho.

Palavras-chave: Meta-heurísticas, Matheurísticas, Problema da Cadeia de Caracteres Mais Próxima, CMSA, Busca Tabu, Auto-adaptação

Abstract

The Closest String Problem (CSP) is a combinatorial optimization problem that seeks to determine a string that best approximates a set of strings with the same length. The CSP is part of a set of optimization problems involving string comparison, with applications in Molecular Biology. This work proposes a Self-Adaptive Construct, Merge, Solve & Adapt (Adapt-CMSA) algorithm using mechanisms from Tabu Search metaheuristic to solve the CSP. CMSA is a matheuristic that explores the problem characteristics to generate a reduced sub-instance of the original problem instance, subsequently employing an exact solver for its resolution. The self-adaptive version of CMSA enhances the algorithm with mechanisms for automatic adaptation of its parameters, making it less sensitive to its configuration across different instances. Experimental results on artificial and real instances demonstrate that the proposed approach significantly outperforms recent heuristic methods proposed in the literature. The optimal value is found in the majority of instances, and the average gap is considerably reduced. Additionally, statistical analyses are conducted to assess the importance of the algorithm's components in its good performance.

Keywords: Metaheuristics, Matheuristics, Closest String Problem, CMSA, Tabu Search, Self-adaptation

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1

Introdução

A Bioinformática é uma área interdisciplinar que emprega técnicas computacionais e estatísticas para manusear e interpretar dados biológicos, abrangendo os campos da ciência da computação, matemática, física e biologia. Segundo Festa [18] e Schatz [56], surge como resultado dos avanços científicos proporcionados pelo sequenciamento do genoma humano com o *Human Genome Project* [62]. Esse marco histórico desencadeou um movimentação global para o sequenciamento genômico de diversas outras espécies, de modo que o conhecimento derivado desses esforços impulsionou avanços significativos tanto na biologia quanto na medicina. Pesquisas em genômica e proteômica comparativas contribuiram para o desenvolvimento de medicamentos e vacinas e para a identificação de doenças genéticas.

O crescimento abrupto da geração de dados genômicos promoveu o início de uma era de ciência de dados biológicos e biomédicos. Coletar novas observações se tornou corriqueiro através de sensores e microscópios digitais de alta resolução, instrumentos de sequenciamento de DNA, sistemas de imagem por ressonância magnética, tomografia computadorizada e dispositivos de monitoramento de sinais vitais. Além disso, com a difusão e popularização da *Web*, esse tipo de dado passou a ser facilmente disponibilizado na internet em grandes bancos de dados disponíveis gratuitamente para a comunidade científica, como o *National Center for Biotechnology Information* (NCBI)[1]. Além dos avanços tecnológicos na coleta, armazenamento e disponibilização desses dados, o desenvolvimento de ferramentas, sistemas, algoritmos e modelagens computacionais tem possibilitado a sua rápida análise e interpretação.

A Pesquisa Operacional (PO) [16] é uma área de conhecimento dedicada ao estudo, desenvolvimento e aplicação de métodos voltados para facilitar a tomada de decisão em diversas áreas de atuação. Através do uso de técnicas de modelagem matemática e algoritmos computacionais, a PO auxilia na análise dos aspectos e particularidades de problemas complexos, permitindo a tomada de decisão e a melhoria de processos. Sendo assim, o crescimento da bioinformática

[1] https://www.ncbi.nlm.nih.gov/

motivou a comunidade de pesquisa operacional a elaborar e propor diversos problemas de otimização combinatória que podem ser formulados a partir de problemas encontrados na biologia molecular.

Um dos marcos fundamentais para a motivação da formulação desses problemas foi a simplificação da estrutura tridimensional do DNA para uma representação unidimensional. Essa representação é composta por uma sequência de caracteres, utilizando um alfabeto de quatro letras — A, C, G e T — que correspondem às quatro bases nitrogenadas Adenina, Citosina, Guanina e Timina, respectivamente. Com base nessa definição, outras estruturas moleculares, como o RNA e as proteínas, puderam ser representadas em formato de *strings*.

Problemas de seleção e comparação de *strings* pertencem a uma classe de problemas em biologia molecular denominada *sequence consensus*, onde o objetivo comum é encontrar uma *string* que melhor represente um conjunto de *strings* recebido como entrada. Essa sequência de caracteres é denominada *consensus* e sua representatividade para com o grupo varia de acordo com o problema, podendo ser proximidade ou distância.

O problema estudado no presente trabalho pertence à classe *sequence consensus* e é denominado Problema da Cadeia de Caracteres Mais Próxima (PCCP), detalhado na Seção 2. Resumidamente, o objetivo do PCCP é encontrar a *consensus* que seja o mais próximo possível do conjunto de *strings*.

1.1 Justificativa

Problemas de comparação e seleção de *strings* surgem na Biologia Molecular em diversos contextos. Festa [18], Kevin Lanctot et al. [26] e Li et al. [30] são alguns dos trabalhos que se dedicaram a explorar e elaborar as diferentes aplicações das *sequences consensus* na Biomedicina.

Um exemplo de aplicação prática é na elaboração de sondas de hibridização (do inglês, *hybridization probes*). Sondas de hibridização são fitas simples de DNA ou RNA utilizadas para procurar por sua sequência complementar em uma determinada amostra de genoma. A formação de uma hibridização estável entre a amostra e a sonda indica a presença do fragmento de gene complementar de interesse. Sondas são utilizadas para distinguir o material genético de diferentes organismos, como vírus [12, 27] e bactérias [24, 48]. Por exemplo, para diagnosticar uma infecção bacteriana, pode-se coletar o material genético de espécies de bactérias patogênicas correlacionadas e encontrar uma sequência de caracteres que se aproxime ao máximo do DNA do grupo. Essa *string* é, portanto, a sonda utilizada para hibridizar com o material genético do possível hospedeiro. Caso a hibridização ocorra, pode-se concluir que pelo menos uma das espécies de bactéria está presente no hospedeiro.

O *design* de iniciadores (ou *primers*) é outra aplicação interessante que está diretamente relacionada com a elaboração de sondas. O *design* de sondas e iniciadores eficientes é um dos

fatores decisivos para o sucesso de uma análise de PCR (*Polymerase Chain Reaction*) para identificação de doenças [27, 31, 36]. O PCR é um método utilizado para amplificar partes de um material genético em várias cópias. Para realizar a amplificação são utilizados os iniciadores, que são pequenas sequências de nucleotídeos complementares a uma região, que servem de ponto de partida para a síntese de DNA. Muitas vezes pode ser necessário desenvolver iniciadores para diferentes organismos ao mesmo tempo, os chamados iniciadores universais, os quais devem ser capazes de hibridizar com o DNA desse conjunto de espécies. Uma hibridização bem sucedida depende, dentre outros fatores, da quantidade de posições em que os nucleotídeos do iniciador e do DNA se complementam (que deve ser a maior possível). Sendo assim, tais iniciadores universais devem representar ao máximo possível o conjunto de espécies para que consigam hibridizar corretamente com todas elas.

Ainda, problemas de *sequence consensus* estão relacionados com o desenvolvimento de medicamentos e vacinas [6, 45, 54]. Supondo que se tenha um conjunto de genes ortólogos de patógenos relacionados, isto é, genes compartilhados por espécies de patógenos irmãos que se modificaram ao longo do tempo, mas que ainda codificam a mesma função. Pode-se encontrar um fragmento de gene que seja o mais conservado possível dentre os genes das espécies, mas não tão presente no DNA do hospedeiro. A descoberta desse fragmento genético possibilita a identificação da informação codificada por ele e o desenvolvimento de uma potencial vacina ou medicamento, o qual afetará apenas o organismo que contém aquele gene.

Todas essas aplicações têm em comum a necessidade de encontrar uma *string* que melhor represente um conjunto de *strings*. Tais problemas são, em sua maioria, computacionalmente intratáveis e, portanto, não são conhecidos algoritmos para encontrar a solução ótima em tempo polinomial. Assim, se faz extremamente relevante o estudo e desenvolvimento de técnicas para encontrar soluções satisfatórias com baixo custo computacional, o que pode ser realizado através do uso de algoritmos aproximativos e heurísticos.

1.2 Objetivos

1.2.1 Objetivo geral

Este trabalho tem como objetivo geral desenvolver e demonstrar a efetividade da matheurística *Construct, Merge, Solve & Adapt* (CMSA) [10] no contexto da resolução do Problema da Cadeia de Caracteres Mais Próxima, visando evoluir o *state-of-the-art* do problema.

1.2.2 Objetivos específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- Implementar a matheurística *Construct, Merge, Solve & Adapt* (CMSA) para resolver o Problema da Cadeia de Caracteres Mais Próxima;

- Aprimorar o algoritmo utilizando estratégias para escapar de ótimos locais e acelerar a convergência;
- Testar o algoritmo desenvolvido em instâncias disponíveis na literatura do problema e compará-lo com algoritmos *state-of-the-art*;
- Realizar testes estatísticos para analisar como os componentes do algoritmo contribuem para o seu bom desempenho;

1.3 Contribuições

O presente trabalho apresenta uma evolução do trabalho entitulado "Uma Abordagem Híbrida CMSA para o Problema da Cadeia de Caracteres mais Próxima" [44], publicado pelo autor no LV Simpósio Brasileiro de Pesquisa Operacional e premiado em 2º lugar como melhor trabalho de Iniciação Científica.

1.4 Organização do Trabalho

O trabalho está organizado da seguinte forma: o Capítulo 2 introduz formalmente o Problema da Cadeia de Caracteres Mais Próxima e apresenta uma breve revisão da literatura do problema. O Capítulo 3 apresenta os fundamentos que norteiam o desenvolvimento deste trabalho. O Capítulo 4 descreve o algoritmo *Adapt*-CMSA com Tabu proposto, detalhando cada uma de suas etapas e especificidades. Por fim, o Capítulo 5 apresenta os experimentos computacionais realizados, resultados obtidos e análises estatísticas.

2

Caracterização do problema

Este capítulo tem como objetivo introduzir o problema abordado neste trabalho, denominado Problema da Cadeia de Caracteres Mais Próxima. A Seção 2.1 o descreve formalmente. A Seção 2.2 apresenta uma formulação matemática para o problema. E, por fim, a Seção 2.3 apresenta uma breve revisão da literatura.

2.1 Problema da Cadeia de Caracteres Mais Próxima

O Problema da Cadeia de Caracteres Mais Próxima (PCCP) é um problema de otimização combinatória que procura determinar uma *string* que seja o *mais próximo possível* de um conjunto finito de *strings* de mesmo comprimento e formadas sobre o mesmo alfabeto. O PCCP faz parte de um conjunto de diversos problemas de otimização que envolvem comparação de *strings*, como *Farthest String Problem*, *Closest Substring Problem*, *Close to Most String Problem* e *Far From Most String Problem*, classificados pela literatura como *Sequences Consensus Problems* [18, 26].

Festa [18] e Kevin Lanctot et al. [26] atribuem ao PCCP a seguinte definição formal:

PROBLEMA DA CADEIA DE CARACTERES MAIS PRÓXIMA

ENTRADA: Conjunto finito $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ de n *strings* de comprimento m e conjunto $\Sigma = \{c_1, c_2, \ldots, c_k\}$ de k caracteres.

OBJETIVO: Encontrar uma sequência $t \in \Sigma^m$, tal que a distância entre t e todas as sequências $s_i \in \mathcal{S}$ seja minimizada.

Nesta definição, cada *string* s_i é formada pelos seguintes caracteres $s_i^1 \cdots s_i^m$.

A Figura 2.1 ilustra de forma didática uma entrada e saída do PCCP. Nesse exemplo é recebido como entrada um conjunto de *strings* $\mathcal{S} = \{$*Homo sapiens, Pan troglodytes, Pongo*

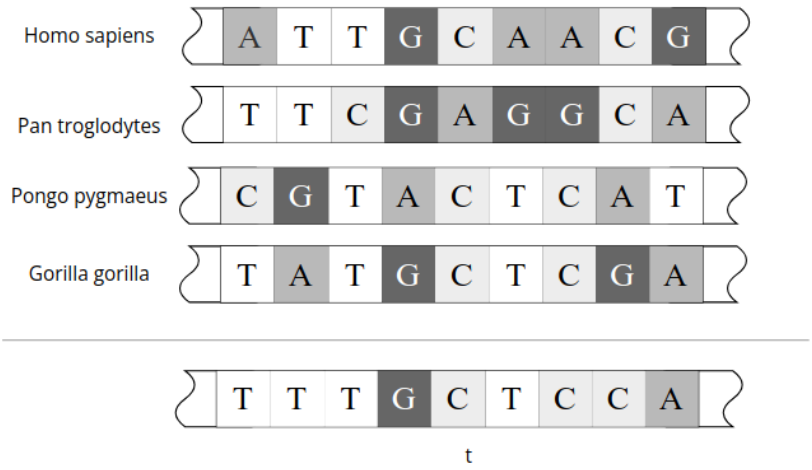


Figura 2.1: Exemplo de instância e solução para o PCCP. Fonte: Autor

pygmaeus, Gorilla gorilla}, representando sequências de DNA de 4 espécies distintas, e é retornada uma *string* t. Todas as *strings* possuem o mesmo comprimento $m = 9$ e são formadas sobre o alfabeto $\Sigma = \{A, T, G, C\}$. Além disso, a distância entre t e cada $s_i \in \mathcal{S}$ deve ser, no máximo, um valor d, que é o mínimo que se pode obter. O cálculo de d depende da métrica de distância utilizada.

Na literatura do PCCP são propostas diferentes métricas para determinar a proximidade entre as *strings*. Por ser amplamente utilizada na literatura [11, 18, 23, 30], neste trabalho é adotada a distância de *hamming*, formalmente definida através da Equação 2.1. De forma resumida, a distância de *hamming* entre duas cadeias de caracteres de mesmo tamanho, s e t, é a quantidade de posições em que diferem.

$$d_H(s,t) = \sum_{i=0}^{|s|} \phi(s_i, t_i) \tag{2.1}$$

Onde $\phi : \Sigma \times \Sigma \rightarrow \{0,1\}$ é definida na Equação 2.2.

$$\phi(a,b) = \begin{cases} 1, & \text{se } a \neq b \\ 0, & \text{caso contrário} \end{cases} \tag{2.2}$$

A Figura 2.2 exemplifica o uso da distância de *hamming* na sequência genética de duas espécies diferentes, *Homo sapiens* e *Pan troglodytes*. Na figura, as sequências diferem nas posições 1 ($\phi(A,T)$), 3 ($\phi(T,C)$), 5 ($\phi(C,A)$), 6 ($\phi(A,G)$) e 7 ($\phi(A,G)$), e, portanto, d_H(*Homo sapiens*, *Pan troglodytes*) = 5, isto é, o somatório dessas disparidades.

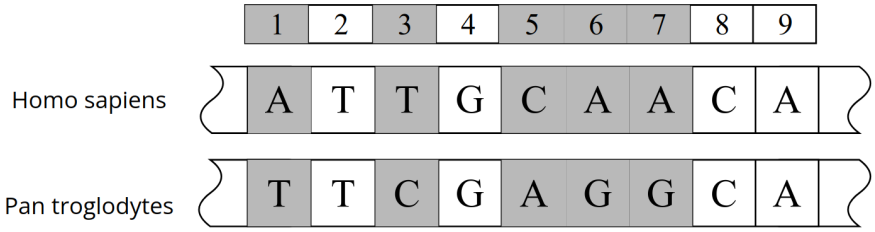


Figura 2.2: *Strings* que possuem distância de *hamming* igual a 5. Fonte: Autor

Uma vez utilizando a distância de *hamming*, o PCCP pode ser resumido na seguinte função objetivo:

$$\min \quad d_H(t, s_i) \leq d, \quad \forall s_i \in \mathcal{S} \tag{2.3}$$

2.2 Formulação matemática

Problemas de otimização são, na maioria das vezes, facilmente representados no formato de formulações matemáticas, facilitando a sua solução por meio de algoritmos de Programação Linear. Meneses et al. [40] propuseram 3 formulações de Programação Linear Inteira distintas para o PCCP, sendo a terceira delas (P3) demostrada por eles como sendo a mais eficiente e, portanto, adotada neste trabalho.

Tome $V_j \subseteq \Sigma$ como o conjunto dos caracteres que aparecem na posição j das *strings* em $\mathcal{S}$. Supondo que o conjunto V_j contenha k caracteres, V_j^k corresponde, portanto, ao k-ésimo caractere de V_j. Define-se uma variável de decisão binária,

$$x_k^j = \begin{cases} 1, & \text{se o caractere } V_j^k \text{ for utilizado em } t_j \\ 0, & \text{caso contrário} \end{cases} \tag{2.4}$$

existente para todo $k \in V_j$ e $j = 1, \ldots, m$.

A formulação matemática para o PCCP é, portanto, definida pelas equações de (2.5) a (2.9).

$$\text{P3:} \quad \min \quad d \tag{2.5}$$

$$\text{s.a} \quad \sum_{k \in V_j} x_k^j = 1 \qquad j = 1, \ldots, m; \tag{2.6}$$

$$m - \sum_{j=1}^{m} x_{s_i^j}^j \leq d \qquad i = 1, \ldots, n; \tag{2.7}$$

$$x_k^j \in \{0, 1\} \qquad j = 1, \ldots, m;\ k \in V_j; \tag{2.8}$$

$$d \in \mathbb{Z}_+ \tag{2.9}$$

As restrições (2.6) garantem que apenas um caractere de V_j seja selecionado para a posição

j. As restrições (2.7) garantem que o cálculo da distância de *Hamming* entre $s_i \in \mathcal{S}$ e t leve em consideração apenas os caracteres de s_i que aparecem em t, e que essa distância seja limitada por d. Por fim, as restrições (2.8) e (2.9) definem o domínio das variáveis x_k^j e d.

2.3 Trabalhos relacionados

Cohen et al. [13], Roman [52] e Struik [58] foram alguns dos primeiros trabalhos a dissertar sobre o *Covering Code Problem*, contextualizado na área de Teoria dos Códigos e Codificação de Ruídos. O objetivo neste problema é encontrar um código binário que melhor descreva os dados a serem transmitidos em um meio de propagação, como forma de garantir que esses dados, mesmo que distorcidos, possam ser entendidos corretamente no destino.

Posteriormente, Frances and Litman [19] provaram que a versão de decisão do *Covering Code* é $\mathcal{NP}$-completo através de uma redução do 3SAT. Em seguida, Ben-Dor et al. [8] se referem à generalização do *Covering Code*, com alfabeto de tamanho variável, como *Consensus Problem*. No trabalho, os autores contextualizam o problema na Biologia Computacional pela primeira vez. Stojanovic et al. [57] estudam a versão parametrizada do problema, com $d = 1$, a qual é denominada *1-Mismatch Problem* e a apresentam no contexto de alinhamento de sequências de DNA. Ainda, os autores propõem um algoritmo polinomial para resolvê-lo à otimalidade.

Kevin Lanctot et al. [26] é o primeiro trabalho a nomear o *Consensus Problem* como Problema da Cadeia de Caracteres Mais Próxima. Nele, os autores estudam as aplicações na Biologia Computacional dos problemas de comparação de *strings*, provam que o PCCP é $\mathcal{NP}$-difícil e propõem um algoritmo polinomial $(\frac{4}{3}+\varepsilon)$-aproximado para qualquer valor pequeno de $\varepsilon > 0$.

A exploração da versão genérica do PCCP com o uso de programação matemática foi alavancada com o trabalho de Meneses et al. [40]. Os autores propuseram 3 formulações de Programação Inteira para o problema e desenvolveram um algoritmo heurístico para gerar podas para um *branch-and-bound*. Posteriormente, no trabalho de Gomes et al. [23], tal algoritmo foi incrementado com a adoção de uma abordagem paralela. Em seguida, Chimani et al. [11] estuda uma das formulações de Meneses et al. [40] (a mesma utilizada no presente trabalho) e propõem uma modificação que reduz a quantidade de variáveis por um fator de $1/|\Sigma|$. Os autores realizam testes extensivos em cerca de 6000 instâncias e concluem que, apesar de ser uma formulação eficiente, a original proposta em Meneses et al. [40] performa melhor.

No campo das meta-heurísticas, a literatura do PCCP possui diversos trabalhos com abordagens diversas para encontrar boas soluções em tempos cada vez menores. Além de Gomes et al. [23], Liu et al. [35] também utilizam uma abordagem paralela em dois algoritmos, um Algoritmo Genético e um *Simulated Annealing*, e as comparam com suas versões sequenciais. Faro and Pappalardo [17] apresentam um algoritmo *Ant Colony*, o qual é demonstrado superar as versões sequenciais dos algoritmos propostos por Liu et al. [35] em instâncias de DNA

de tamanho pequeno e moderado. Julstrom [25] apresenta três versões distintas de algoritmos genéticos utilizando diferentes heurísticas de inicialização da população.

Há alguns trabalhos, ainda, que apostam numa abordagem híbrida entre algoritmos heurísticos distintos. Liu et al. [33], por exemplo, propõem uma híbridização entre Algoritmo Genético e *Simulated Annealing*. Já Babaie and Mousavi [7] desenvolvem um Algoritmo Memético, o qual combina busca local com um Algoritmo Evolucionário, e mostram superar as abordagens de Julstrom [25] e Liu et al. [33]. Em Liu et al. [34] os autores adicionam uma etapa de busca local ao algoritmo aproximativo polinomial *Largest Distance Decreasing* (LDDA) proposto por eles em [32].

Pappalardo et al. [47], por sua vez, potencializam um algoritmo *Simulated Annealing* com uma abordagem *Greedy Walk* (SAGW), uma heurística construtiva gulosa para gerar boas soluções iniciais. No trabalho, o SAGW é comparado com os algoritmos meta-heurísticos mais recentes da literatura até então e os autores concluem que o algoritmo performa melhor que as abordagens de Babaie and Mousavi [7], Faro and Pappalardo [17], Julstrom [25], Liu et al. [33] e Liu et al. [34] nos grupos de instâncias testados.

Por fim, um dos trabalhos mais recentes encontrados na literatura é o de Santos [55]. Nele, o autor propõe 3 algoritmos diferentes: *Simulated Annealing* (SA), *Variable Neighborhood Search* (VNS) e *Hybrid Discrete Particle Swarm Optimization* (HDPSO). Os experimentos computacionais são realizados utilizando as instâncias de Chimani et al. [11] e demonstram uma superioridade em qualidade de solução do HDPSO em relação aos algoritmos SA e VNS.

3

Fundamentação teórica

Este capítulo tem como objetivo introduzir a teoria que serve de fundamento para o presente trabalho. A Seção 3.1 introduz a teoria da $\mathcal{NP}$-completude. A Seção 3.2 apresenta conceitos importantes em otimização combinatória. As Seções 3.3, 3.4 e 3.5 se debruçam sobre abordagens exatas, heurísticas e híbridas, respectivamente.

3.1 NP-completude

A teoria da $\mathcal{NP}$-completude foi fundamentada no trabalho "*The Complexity of Theorem Proving Procedures*" de Stephen Cook [14] em 1971. Nele, é introduzido o conceito de problemas $\mathcal{NP}$-completos e demonstrada a $\mathcal{NP}$-completude do SAT (*Boolean Satisfiability Problem*). Desde então, a comunidade teórica de computação tem empenhado esforços na discussão de $\mathcal{P}$ *versus* $\mathcal{NP}$, de modo que a prova da intratabilidade de problemas $\mathcal{NP}$-completos é uma das questões em aberto mais intrigantes na matemática e ciência da computação.

Um problema é considerado intratável quando é tão difícil que nenhum algoritmo de complexidade polinomial pode resolvê-lo. Em outras palavras, a complexidade de um algoritmo para resolver tal problema aumenta exponencialmente com o tamanho da entrada, tornando impraticável encontrar uma solução em um tempo razoável para instâncias de tamanho considerável. Nesse contexto, são definidas as classes de complexididade de problemas $\mathcal{P}$ e $\mathcal{NP}$.

A classe $\mathcal{P}$ abrange os problemas que possuem algoritmos de complexidade polinomial para resolvê-los, isto é, problemas cujo tempo para resolver é $O(p(n))$, em que n é o tamanho da entrada e p uma função polinomial. Para alguns problemas, no entanto, apenas são conhecidos algoritmos exponencias, os quais, em geral, são algoritmos de enumeração/busca exaustiva.

Por sua vez, a classe $\mathcal{NP}$ engloba problemas que possuem um algoritmo de tempo polinomial para verificar uma solução. A verificação é feita com base em uma instância e um certificado, e consiste de checar se o certificado é válido para a instância fornecida. Por exemplo, tome a versão de decisão do PCCP, onde se quer saber se existe uma *string* t de tamanho

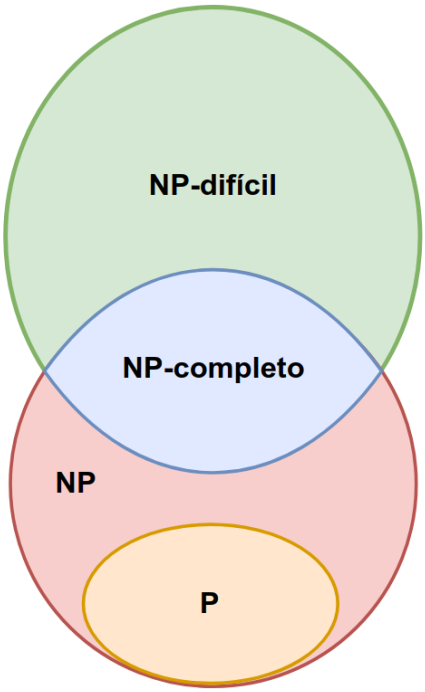


Figura 3.1: Relação entre as classes de problemas se $\mathcal{P} \neq \mathcal{NP}$. Fonte: Autor

m, cuja distância máxima às *strings* de um conjunto S é igual a um valor d. Suponha uma instância, contendo o alfabeto Σ e o conjunto de *strings* $\mathcal{S}$, tal que $s_i \in \Sigma^m \ \forall s_i \in \mathcal{S}$. Um certificado é, portanto, constituído pela *string* t e a maior distância d, representando uma solução para o problema. Verificar tal certificado consiste em confirmar que t possui comprimento m e que $t_i \in \Sigma$, para todas as posições $i \in 1, \ldots, m$. Além disso, a distância entre t e cada *string* $s_i \in \mathcal{S}$ deve ser, no máximo, d. Portanto, o PCCP é um problema em $\mathcal{NP}$. Ainda, vale ressaltar que $\mathcal{P} \subseteq \mathcal{NP}$ e, portanto, todo algoritmo facilmente resolvível também é facilmente verificável.

Há, ainda, a classe de problemas $\mathcal{NP}$-completos. Estes possuem a propriedade de que qualquer problema em $\mathcal{NP}$ pode ser reduzido polinomialmente a eles. Dessa forma, os problemas $\mathcal{NP}$-completos são os mais difíceis em $\mathcal{NP}$. A $\mathcal{NP}$-completude é uma propriedade de problemas de decisão, isto é, problemas para os quais a resposta é "sim" ou "não". Problemas de outra natureza, como de otimização e busca, que são tão difíceis quanto os problemas $\mathcal{NP}$-completos pertencem à classe $\mathcal{NP}$-difícil. Esta abrange todos os problemas, pertencentes ou não à classe $\mathcal{NP}$, para os quais um problema $\mathcal{NP}$-completo pode ser reduzido. Dessa forma, todo problema de decisão $\mathcal{NP}$-completo é também $\mathcal{NP}$-difícil, bem como suas versões de otimização e busca. A Figura 3.1 ilustra a relação das classes de complexidade $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-completo e $\mathcal{NP}$-difícil, caso $\mathcal{P} \neq \mathcal{NP}$.

Resolver qualquer problema $\mathcal{NP}$-completo em tempo polinomial confirma que $\mathcal{P} = \mathcal{NP}$. No entanto, a maioria dos pesquisadores da teoria da computação acredita que $\mathcal{P} \neq \mathcal{NP}$. Até o momento nenhum algoritmo de tempo polinomial para um problema $\mathcal{NP}$-completo foi descoberto, fortalecendo essa convicção. Sendo assim, $\mathcal{P}$ *versus* $\mathcal{NP}$ se mantém como um dos

tópicos mais intrigantes e desafiadores da computação.

3.2 Otimização Combinatória

Muitos problemas matemáticos envolvem a busca pelo melhor valor ou conjunto de valores, de modo que a definição de melhor está diretamente relacionada com o problema em questão. Tais problemas, conhecidos como problemas de otimização, encontram aplicações em inúmeros contextos e são o objeto de estudo de diferentes áreas, como Pesquisa Operacional, Inteligência Artificial e Controle de Sistemas.

Formalmente, segundo Papadimitriou and Steiglitz [46], um problema de otimização é um par (D, f), tal que D representa o domínio e f uma função de custo $f : D \to \mathbb{R}$. O objetivo do problema é encontrar um elemento x^* em D de modo que $f(x^*) \leq f(x)$ (problema de minimização) ou $f(x^*) \geq f(x)$ (problema de maximização) para todo $x \in D$. Nesse caso, x^* é o chamado ótimo global ou solução ótima. O domínio D é, portanto, um conjunto de soluções factíveis, contendo todas as possibilidades de elementos que podem constituir uma solução válida. f, por sua vez, é a função objetivo do problema, responsável por avaliar a qualidade das soluções de acordo com o problema em questão.

Os problemas de otimização podem ser classificados entre contínuos e discretos. Nos problemas contínuos, D é formado por todo o conjunto de números reais, e um elemento x_0 qualquer nesse domínio é um valor ou conjunto de valores nos reais. Já nos discretos, D é um conjunto finito formado por variáveis inteiras, grafos ou conjuntos e x_0 é, portanto, uma variável, um grafo ou um conjunto. Estes são os chamados problemas combinatórios.

Nesse contexto, surge uma terminologia importante a se introduzir. Definimos uma *instância* como os dados recebidos como entrada que possibilitam a construção de uma solução. O par (D, f) é uma instância. Ainda, um *algoritmo* para um problema de otimização é dito *exato* se sempre retorna uma solução ótima para qualquer instância recebida como entrada. Tal algoritmo é considerado *eficiente* se essa solução ótima é encontrada em tempo polinomial no tamanho da instância.

Dado que encontrar a solução ótima para um problema pode ser computacionalmente intratável, como descrito na Seção 3.1, é normalmente mais fácil encontrar soluções que são as melhores em um pedaço do espaço de busca, os chamados *ótimos locais*. A *vizinhança* $N(x_0)$ de uma solução x_0 qualquer são as soluções, de alguma forma, próximas a ela. Vizinhanças podem ser definidas de diferentes formas e dependem do problema em questão. Um ponto x_0 é denominado *mínimo local* se $f(x_0) \leq f(x)\ \forall x \in N(x_0)$ e *mínimo global* se $f(x_0) \leq f(x)\ \forall x \in D$. Por outro lado, x_0 é denominado *máximo local* se $f(x_0) \geq f(x)\ \forall x \in N(x_0)$ e *máximo global* se $f(x_0) \geq f(x)\ \forall x \in D$. A Figura 3.2 exemplifica pontos de máximo e mínimo globais e locais de uma função em um cenário de otimização contínua. A busca por ótimos locais é uma tarefa comum em algoritmos de busca local, descritos posteriormente na Seção 3.4.

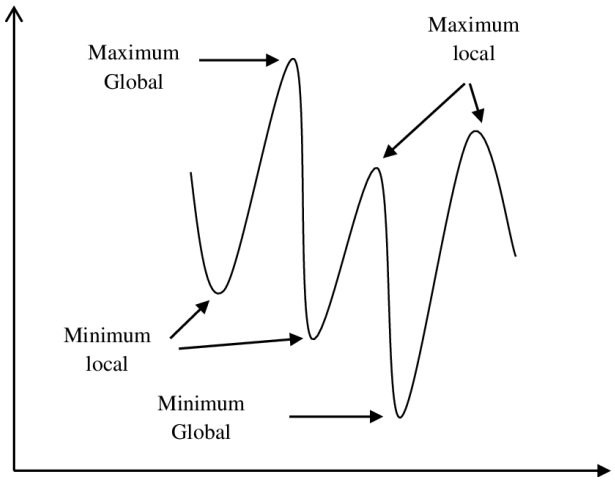


Figura 3.2: Ótimos locais e globais de uma função. Fonte: Meraihi [42]

3.3 Métodos exatos

Algoritmos de otimização podem ser classificados entre exatos e heurísticos. Metodos exatos, em geral, são preferidos para resolver problemas de otimização pertencentes a classe $\mathcal{P}$, isto é, cujo esforço computacional cresce polinomialmente no tamanho da entrada. Isto porque, como descrito na Seção 3.1, algoritmos exatos conhecidos para problemas $\mathcal{NP}$-difícil são exponenciais e, para instâncias de grandes proporções, torna-se impossível encontrar a solução ótima rapidamente. Para estes problemas são utilizadas abordagens heurísticas, a serem descritas na Seção 3.4. Nesta seção serão abordados alguns métodos exatos, como Programação Linear e Algoritmos Enumerativos.

3.3.1 Programação Linear

Uma função linear é definida como

$$f(x) = c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{3.1}$$

em que $x = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ são as variáveis contínuas e $c = (c_1, c_2, \ldots, c_n) \in \mathbb{R}^n$ um conjunto de constantes. Um problema é dito linear se a função objetivo depende linearmente das variáveis de decisão e se a relação entre todas as variáveis é linear. Tais relações são expressadas no formato de restrições, que podem ser de equalidade $f_i(x) = b_i$ ou de inequalidade $f_i(x) \geq b_i$ e $f_i(x) \leq b_i$, de modo que f_i são funções lineares. Para fins de simplificação de representação, o conjunto de restrições pode ser representado em formato matricial. Suponha um problema com m restrições e n variáveis. Tomando A como uma matriz $m \times n$ e b um vetor de dimensão n, as restrições de equalidade e inequalidade são representadas por $Ax = b$ e $Ax \geq b$ ou $Ax \leq b$, respectivamente.

A formulação de um problema de otimização linear é, portanto, constituída de função obje-

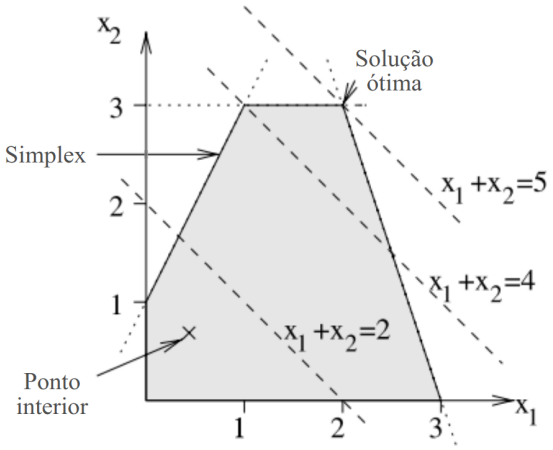


Figura 3.3: Polígono formado pelas restrições de um problema de programação linear. Fonte: Rothlauf [53] (Adaptado)

tivo e restrições. Uma formulação é dita estar na *forma canônica* se for da forma

$$\begin{aligned} \max \quad & c^T x \\ \text{sujeito a} \quad & Ax \geq b \\ & x_i \geq 0 \qquad i = 1, \ldots, n; \end{aligned} \tag{3.2}$$

ou na *forma padrão* se estiver no formato

$$\begin{aligned} \max \quad & c^T x \\ \text{sujeito a} \quad & Ax = b \\ & x_i \geq 0 \qquad i = 1, \ldots, n; \end{aligned} \tag{3.3}$$

A representação de um problema na forma padrão permite a sua solução por meio do método *Simplex*. Em linhas gerais, o *simplex* é um algoritmo que resolve problemas de programação linear (PL) utilizando as restrições para delimitar um politopo no espaço formado pelas n variáveis. A região delimitada pelo politopo é denominada *simplex* e agrupa os pontos no espaço que satisfazem todas as restrições e são, portanto, soluções viáveis. O algoritmo *simplex* consiste de analisar todos os vértices do politopo e encontrar aquele que retorna o melhor valor da função objetivo. A Figura 3.3 exemplifica a construção do polígono (*simplex*) a partir de uma formulação de PL de 2 variáveis, x_1 e x_2, e 5 restrições. O algoritmo analisa os 5 vértices do polígono à procura da solução ótima, que é o ponto (2,3).

Quando todas as variáveis de um problema precisam ser inteiras, este é denominado problema de Programação Linear Inteira (PLI). Neste caso, a formulação padrão apresentada na Equação 3.3 é modificada apenas na restrição de domínio, a qual é substituída por $x_i \in \mathbb{N}$. Diferente de PL, soluções viáveis para um problema PLI são os pontos inteiros dentro dos limites do politopo. Desse modo, o método *simplex* não pode ser utilizado para resolvê-lo, pois os

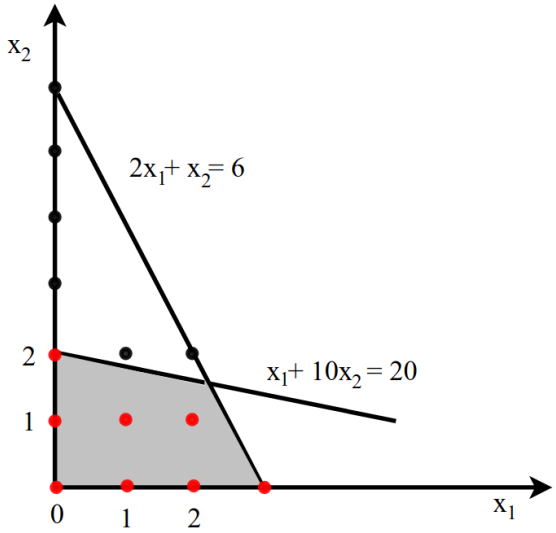


Figura 3.4: Polígono formado pelas restrições de um problema de programação linear inteira. Fonte: Autor

pontos ótimos não ocorrem necessariamente nos vértices. A Figura 3.4 exemplifica o polígono formado a partir de 4 restrições sobre as variáveis x_1 e x_2 de um problema. Os pontos inteiros dentro dos limites do polígono são as soluções viáveis para o problema.

Se as restrições de domíno das variáveis forem removidas, a formulação de um problema de PLI é transformada em PL e, assim, o *simplex* pode ser utilizado para resolvê-lo. Esse processo é denominado *relaxação linear*. No entanto, a solução encontrada pelo *simplex* pode ser contínua e, portanto, não viável para o problema. Nesse caso, tal solução pode ser utilizada como limite inferior (para problemas de minimização) do valor ótimo, servindo de ponto de partida para um outro algoritmo, como o *branch-and-bound*, por exemplo.

3.3.2 Algoritmos Enumerativos

Backtracking

O *backtracking* é um algoritmo de busca exaustiva que opera construindo soluções de forma incremental e recursiva, resultando em uma árvore de enumeração em que cada nó representa um subproblema a ser resolvido durante a busca pela solução ótima. Em cada suproblema, o algoritmo precisa escolher um componente para fazer parte da solução. Caso uma determinada escolha não leve a uma solução viável, o algoritmo retrocede (*backtracks*) e explora outras alternativas. Essa abordagem continua até que todas as possibilidades tenham sido examinadas, e por essa razão, a complexidade do algoritmo é exponencial para problemas de otimização combinatória. No entanto, é uma abordagem eficaz em problemas onde é possível podar ramos da árvore de enumeração para reduzir o número de soluções a serem exploradas.

A Figura 3.5 ilustra a construção de uma árvore de enumeração para o PCCP. Nesse caso, a árvore representa a construção da *string* t, de modo que cada nó representa a decisão de colocar

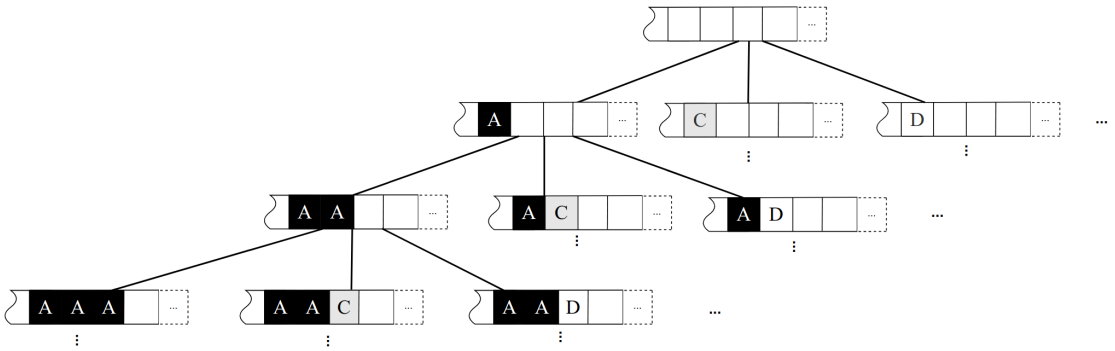


Figura 3.5: Árvore de enumeração gerada pelo *backtracking* para resolver o PCCP. Fonte: Autor

um caractere do alfabeto em uma determinada posição. Sendo assim, a raiz da árvore é a *string* t vazia. Em seguida, o primeiro nível contém os subproblemas associados à primeira posição de t, de modo que cada subproblema é constituído pela escolha de um caractere diferente. O nível seguinte, por sua vez, contém os subproblemas gerados pela solução do subproblema do nível anterior. Assim, se percorrermos a árvore no sentido mais à esquerda, temos no nível 1 a construção da *string* "A", no nível 2 "AA" e no nível 3 "AAA". O procedimento se repete até que as m posições de t sejam preenchidas e a distância de *Hamming* possa ser avaliada. Sempre que é encontrada uma *string* com maior distância de *Hamming* menor que a melhor solução até então é feita a atualização da melhor.

Branch-and-bound

Branch-and-bound [29] é um método enumerativo inteligente que utiliza estratégias de poda para não percorrer todo o espaço de busca. O algoritmo consiste em particionar recursivamente o espaço de busca em subproblemas, um processo denominado "*branching*", e podando ("*bounding*") ramos infrutíferos. Assim como no *backtracking*, a execução do algoritmo pode ser visualizada como a construção de uma árvore, onde cada nó representa um subproblema, a partir do qual se ramificam outros subproblemas obtidos a partir dele. O grande diferencial do *branch-and-bound* são seus mecanismos de redução do espaço de busca, de modo que cada subproblema é avaliado e ramificado apenas se constituir uma solução promissora.

O *branch-and-bound* é frequentemente utilizado para resolver problemas de PLI. Nesse caso, subproblemas são obtidos adicionando novas restrições às variáveis de decisão. Segundo Rothlauf [53], o *branch-and-bound* é aplicado a problemas de PLI da seguinte forma:

1. Inicialmente, o problema é relaxado e resolvido por programação linear, processo descrito na Subseção anterior. Esta solução serve como limite (superior ou inferior) para o valor ótimo da função objetivo. Se o problema for de maximização, este limite é superior, e se for de minimização, inferior. Caso a solução seja inteira, então o problema já está resolvido.

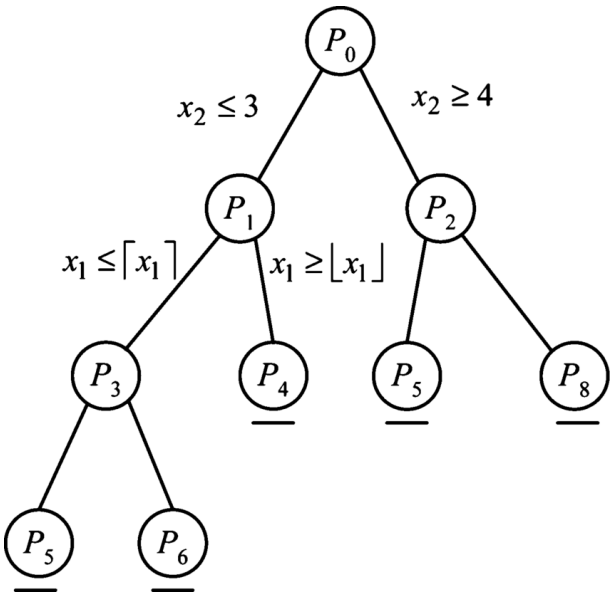


Figura 3.6: Ramificação de um nó na árvore de enumeração do *branch-and-bound*. Fonte: Yu and Lim [63]

2. Em seguida, a construção da árvore se inicia ramificando uma variável de decisão x_i não inteira qualquer. A partir deste nó são geradas duas ramificações: $x_i \leq \lfloor x_i \rfloor$ e $x_i \geq \lceil x_i \rceil$. Quando estes dois subproblemas são resolvidos, retornam recursivamente limites para cada ramificação.

   A Figura 3.6 exemplifica o processo de *branching*. A raíz da árvore representa a decisão sobre a variável x_2 e resulta na sua ramificação em dois subproblemas: $x_2 \leq 3$ e $x_2 \geq 4$. Em seguida, para cada subproblema gerado têm-se a decisão sobre uma outra variável, nesse caso x_1. O processo se repete para todas as variáveis, no pior caso, ou até que o subproblema possa ser descartado.

3. Um subproblema é podado nas seguintes situações:

   - Todas as variáveis de decisão são inteiras. No caso de um problema de minimização, se o valor da função objetivo nesse nó for menor que o limite superior, então este é substituído pelo valor do nó (poda por otimalidade).
   - O subproblema não tem solução viável (poda por inviabilidade).
   - A função objetivo da relaxação do subproblema é pior do que a do limite, ou seja, maior que a do limite superior ou menor que a do limite inferior (poda por limite).

4. O algoritmo continua expandindo subproblemas, escolhendo o próximo a ser ramificado de acordo com algum critério, e retorna recursivamente sempre que um nó é podado.

A implementação de um algoritmo *branch-and-bound* pode variar na escolha da estratégia de ramificação dos nós, na ordem de exploração da árvore e nas regras de poda. Além disso,

embora apresentado sob o contexto de problemas PLI, *branch-and-bound* também pode ser aplicado a problemas não linares.

Visando facilitar a aplicação do *branch-and-bound* à resolução de diferentes problemas de otimização, diversos pacotes de *software* de otimização foram desenvolvidos. Dentre eles, destacam-se CPLEX[1], Gurobi[2] e LINDO[3].

3.4 Métodos heurísticos

Métodos heurísticos são preferíveis para resolver problemas de otimização quando é inviável explorar a estrutura matemática do problema. Heurísticas utilizam conhecimentos específicos do problema para encontrar boas soluções a um baixo custo computacional. As meta-heurísticas (MHs), por sua vez, são heurísticas de propósito geral, isto é, requerem poucas modificações para serem aplicadas a um problema específico. A principal característica das MHs é a utilização de estratégias de exploração do espaço de busca e fuga de ótimos locais através de mecanismos de intensificação e diversificação.

Ao longo dos anos diversas meta-heurísticas foram propostas e aplicadas aos mais diferentes problemas. Dessa forma, foram cunhadas diversas classes para caracterizar e agrupar tais algoritmos. A Figura 3.7 apresenta a classificação das meta-heurísticas mais populares da literatura. As duas classes mais genéricas que classificam grande parte das MHs são os algoritmos *baseados em população* e *baseados em trajetória*. Os algoritmos *baseados em população* exploram o espaço de busca a partir de um conjunto finito de soluções candidatas, as quais são constantemente melhoradas e eventualmente utilizadas para gerar novas soluções ao longo das iterações. Exemplos desse tipo de meta-heurística são Algoritmos Genéticos, *Particle Swarm Optimization* e *Ant Colony*. Já os algoritmos *baseados em trajetória* buscam por uma solução ótima melhorando iterativamente uma única solução corrente. *Variable Neighborhood Search*, *Iterated Local Search*, *Greedy Randomized Adaptive Search Procedure* (GRASP), Tabu *Search* e *Simulated Annealing* pertencem a essa classe.

As próximas subseções focam nas vertentes de algoritmos heurísticos importantes para o desenvolvimento do presente trabalho, sendo elas as heurísticas construtivas e a busca local.

3.4.1 Heurísticas Construtivas

Heurísticas construtivas consistem de construir soluções de forma iterativa e incremental. A partir de uma solução vazia, adicionam componentes até que uma solução completa seja formada. A seleção desses componentes depende da avaliação de uma função heurística que determina o quanto o componente agrega à qualidade da solução.

[1] IBM ILOG CPLEX Optimization Studio
[2] Gurobi Optimizer
[3] LINDO Software

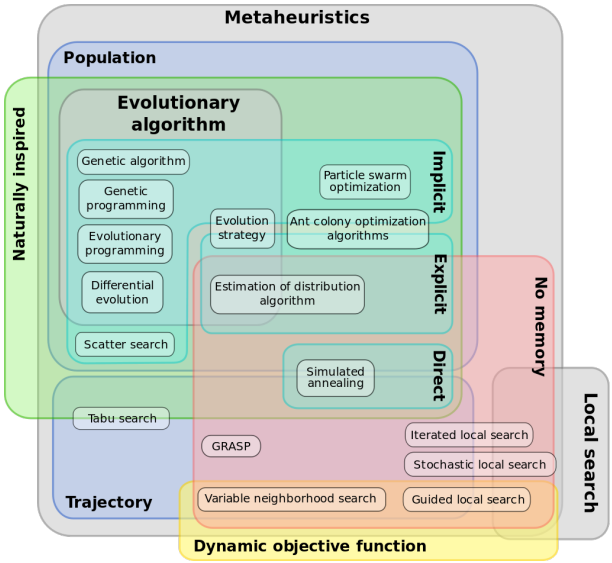


Figura 3.7: Classificação das meta-heurísticas. Fonte: Wikipedia

Uma abordagem construtiva muito comum é a gulosa. Nela, para cada etapa da construção é escolhido o componente que retorna o melhor valor da função heurística. Essa abordagem é denominada gulosa, porque a escolha é realizada localmente de forma ótima e não leva em consideração o seu impacto no futuro. Em geral, soluções construídas de forma gulosa são melhores que as geradas de forma aleatória, no entanto são limitadas em diversidade, de modo que conseguem gerar poucas soluções distintas.

Algumas heurísticas construtivas gulosas foram propostas na literatura do PCCP. Pappalardo et al. [47], por exemplo, propõem uma heurística *Greedy Walk* que escolhe, para cada posição j da *string* t, o caractere s_i^j de uma *string* $s_i \in \mathcal{S}$ para o qual a distância de *Hamming* até a posição j é a maior. Já [7] constrói uma *string* selecionando, para cada posição j, o caractere com maior frequência na posição j das *strings* de entrada.

Os resultados produzidos por heurísticas construtivas não são garantidamente ótimos, no entanto oferecem um bom ponto de partida para algoritmos de busca local. Diversas meta-heurísticas possuem uma fase construtiva seguida de busca local, dentre elas GRASP e *Ant Colony*.

3.4.2 Busca Local

A busca local é um dos métodos heurísticos mais simples e populares existente na solução de problemas de otimização combinatória. De modo geral, consiste em explorar as vizinhanças de uma solução corrente s à procura de uma melhor. A vizinhança $N(s)$ de uma solução s é formada por todas as soluções do espaço de busca que podem ser obtidas ao aplicar modificações em s. Sempre que o algoritmo encontra um vizinho melhor, este passa a ser a nova solução corrente s. A busca termina quando encontra-se um ótimo local, isto é, quando nenhuma vizinhança de s possui valor da função objetivo melhor que $f(s)$.

Existem duas estratégias comuns de exploração das vizinhanças de uma solução: *best improvement* e *first improvement*. Na estratégia *best improvement*, o algoritmo analisa todos os vizinhos de s e se desloca para o melhor dentre eles. Essa operação pode ser custosa a depender do tamanho da vizinhança de s, no entanto provê mais chances de encontrar uma boa solução. Já a abordagem *first improvement* se desloca para o primeiro vizinho de s que melhora o valor da função objetivo, evitando, portanto, explorar toda a sua vizinhança. Sendo assim, os resultados de uma estratégia *first improvement* podem ser piores que os de *best improvement*, no entanto requerem menos tempo computacional para serem alcançados.

Algoritmos puros de busca local costumam ficar presos em ótimos locais, impedindo que explorem ao máximo o espaço de busca e encontrem soluções mais próximas do ótimo global. Sendo assim, diversas meta-heurísticas foram propostas com objetivo de incrementar a busca local com mecanismos de diversificação. De particular relevância para este trabalho é a meta-heurística *Tabu Search*, um algoritmo *baseado em trajetória* com o uso de *memória*.

Tabu *Search* [21, 22, 28] é uma busca local inteligente que utiliza uma memória de curto prazo para guiar o processo de transformação da solução corrente s. Listas tabu são estruturas de dados que armazenam elementos, combinações ou movimentos proíbidos de serem realizados na modificação da solução. Assim, o algoritmo evita ficar repetindo passos que foram realizados em um passado recente e não ocasionaram em melhora na função objetivo. A utilização da lista tabu na busca local é uma estratégia de diversificação, forçando o algoritmo a explorar regiões do espaço de busca não visitadas ainda. Em geral, define-se um tempo de detenção (ou *tabu tenure*) para os elementos na lista tabu, de modo que estes tem sua proibição anulada após uma quantidade de iterações determinada. Ainda, podem ser estabelecidas ocasiões em que uma regra na lista tabu pode ser ignorada, os chamados critérios de aspiração (*aspiration criteria*). O critério mais comum é o de permitir que um atributo na lista tabu seja utilizado para modificar s se este melhorar o valor da função objetivo. Essa abordagem tem como objetivo evitar que o algoritmo fique preso em soluções subótimas.

O Algoritmo 1 apresenta o pseudocódigo de uma implementação básica do Tabu *Search*. O núcleo do algoritmo é o *loop* principal, que consiste de repetidamente tentar melhorar a solução atual s analisando a sua vizinhança $\mathcal{N}(s)$. $\mathcal{N}(s)$, nesse caso, representa o conjunto de vizinhos de s que podem ser gerados a partir de movimentos não proibidos pela lista tabu T. Sempre que um movimento não melhorar a função objetivo f, este é adicionado em T.

3.5 Métodos híbridos

Além de abordagens puramente exatas ou heurísticas, vários autores têm explorado o desenvolvimento de métodos que combinam diferentes componentes algorítmicos para resolver problemas, dando origem à vertente de algoritmos híbridos. Este termo abrange qualquer tipo de combinação entre algoritmos distintos, mas é frequentemente usado para descrever a colabora-

Algoritmo 1 Pseudocódigo do algoritmo Tabu *search*

```
1:  procedure TABU SEARCH
2:      Construa uma solução inicial s_0
3:      s := s_0, s* := s_0, f* := f(s_0), T := ∅
4:      while critério de parada não satisfeito do
5:          s := arg min_{s'∈N(s)} f(s')
6:          if f(s) < f* then
7:              f* := f(s), s* := s
8:          else
9:              Adicione o movimento realizado em T
10:         end if
11:     end while
12: end procedure
```

ção entre métodos exatos e meta-heurísticos.

Puchinger and Raidl [50] categorizam métodos híbridos em colaborativos e integrativos. No primeiro tipo, os diferentes métodos são executados independentemente, seja sequencialmente ou em paralelo, e trocam informações entre si. No segundo tipo, um dos métodos atua como mestre e coordena chamadas para algoritmos subordinados, de modo que estes funcionam como componentes embutidos do algoritmo mestre. A combinação adequada de algoritmos exatos e meta-heurísticas pode se beneficiar da sinergia e apresentar desempenho significativamente superior em termos de qualidade da solução e tempo computacional em comparação com a utilização isolada de cada técnica.

A hibridização entre algoritmos exatos e heurísticos tem sido bastante explorada nos últimos anos devido ao seu potencial performático em comparação com o uso de técnicas puras [1, 9, 38, 50, 61]. Um campo que tem ganhado destaque é o das "*matheurísticas*", as quais combinam técnicas de programação matemática com meta-heurísticas. Em particular, um algoritmo matheurístico que tem se destacado é o Construct, Merge, Solve & Adapt (CMSA), proposto para o Problema da Cadeia de Caracteres Mais Próxima (PCCP) no presente trabalho e descrito em seu formato geral a seguir.

3.5.1 Construct, Merge, Solve & Adapt

O algoritmo *Construct, Merge, Solve & Adapt* (CMSA) foi proposto por Blum et al. [10] como um algoritmo genérico para problemas de otimização combinatória. Pertence à classe de meta-heurísticas híbridas, na qual os algoritmos combinam diferentes técnicas, heurísticas e exatas, para resolver problemas.

A ideia principal do algoritmo é explorar características particulares do problema em questão e transformá-lo em um problema reduzido (não necessariamente o mesmo), no qual se possa utilizar um algoritmo exato para resolvê-lo. Idealmente, deve-se obter um problema reduzido que possa ser resolvido de forma exata em instâncias grandes com um baixo tempo computacional em relação ao problema original. Assim, o CMSA pode tirar proveito de *solvers* disponíveis na literatura, possibilitando sua aplicação a problemas de grande escala de maneira mais efici-

ente.

O algoritmo CMSA consiste em quatro principais etapas que são executadas a cada iteração. A primeira etapa, `Construção`, cria um conjunto de componentes que formarão uma solução S. Em seguida, na etapa `Combinação`, diferentes componentes são mesclados em um único conjunto de componentes C'. Após a fusão, entra em ação a etapa `Solução`, que tenta combinar os diferentes componentes de forma a obter uma solução válida S'_{opt} para o problema utilizando um método exato. Por fim, a etapa `Adaptação` elimina componentes de C' que não são utilizados na melhor solução por um determinado número de iterações. Essa etapa visa otimizar o desempenho do algoritmo, removendo componentes que não contribuem para a solução final.

O Algoritmo 2 apresenta um pseudocódigo genérico para o CMSA. O procedimento recebe como entrada a instância I, a quantidade de soluções n_a a serem geradas por iteração e a idade máxima $idade_{\max}$ permitida para um componente permanecer no conjunto C'. Por fim, o algoritmo retorna a melhor solução encontrada, S_{bsf}.

Primeiramente, nas linhas 2 e 3, as estruturas de dados são inicializadas. É importante destacar o vetor `idade`, que registra há quantas iterações um componente está no conjunto de componentes C'. Em seguida, a etapa `Construção` é executada na linha 6, gerando um conjunto de componentes S. Para cada componente, sua `idade` é definida como 0 na linha 8. A etapa `Combinação` consiste em unir os conjuntos de componentes (linha 9). Já a etapa `Solução`, na linha 12, busca a melhor combinação possível de componentes de C' para formar uma solução viável para o problema, com o menor custo entre as opções disponíveis. Note que o problema resolvido em `Solução` não é necessariamente o problema original. Por fim, a etapa `Adaptação`, na linha 15, remove do conjunto C' os componentes que estão presentes há mais de $idade_{\max}$ iterações sem contribuir para a melhor solução atual. Vale ressaltar que todos os componentes contidos em S'_{opt} têm `idade` 0.

Algoritmo 2 Pseudocódigo do CMSA

```
 1: procedure CMSA(I, n_a, idade_max)
 2:     S_bsf := ∞, C' := ∅
 3:     idade[c] := 0 para todo c ∈ C
 4:     while critério de parada não alcancado do
 5:         for i = 1, ..., n_a do
 6:             S:= GeradorDeSoluções(C)                          ▷ Construção
 7:             for all c ∈ S and c ∉ C' do
 8:                 idade[c] := 0
 9:                 C' := C' ∪ {c}                                ▷ Combinação
10:             end for
11:         end for
12:         S'_opt := SolverPLI(C')                               ▷ Solução
13:         if S'_opt é melhor que S_bsf then S_bsf := S'_opt
14:         end if
15:         Adaptação(C', S'_opt, idade_max)                      ▷ Adaptação
16:     end while
17:     return S_bsf
18: end procedure
```

Self-Adaptive CMSA

No recente trabalho de Akbay et al. [5] uma versão auto-adaptativa do CMSA foi proposta com o objetivo de tornar o algoritmo menos sensível aos seus parâmetros em diferentes instâncias. A ideia principal do *self-adaptive* CMSA é ter uma quantidade fixa de parâmetros que se ajustam dinamicamente ao longo da execução.

O pseudocódigo do *self-adaptive* CMSA é apresentado no Algoritmo 3. Diferente da versão original do CMSA, o *self-adaptive* recebe como entrada 5 hiperparâmetros, além da instância I: t_{prop}, t_{ILP}, α_{LB}, α_{UB} e α_{red}. Estes servem como referência para o ajuste dos parâmetros auto-adaptativos n_a e α_{bsf}. O parâmetro n_a tem a mesma função que na versão padrão do CMSA. Aqui, no entanto, se autoajusta ao longo das iterações para controlar a quantidade de componentes gerados. Já α_{bsf} ($0 \leq \alpha_{bsf} \leq 1$) regula quão parecidas com a melhor solução S_{bsf} são as novas soluções geradas pelo procedimento $GeradorDeSolucoes(C, S_{bsf}, \alpha_{bsf})$. Os hiperparâmetros α_{LB} e α_{UB} servem como limites inferior e superior, respectivamente, para α_{bsf}. A ideia é que quão maior for o valor de α_{bsf} maior é a similaridade das novas soluções com S_{bsf}.

O algoritmo inicia sua execução na linha 2 gerando uma solução inicial, a qual inicializa a melhor solução corrente S_{bsf}. Em seguida, na linha 3, são inicializados os parâmetros auto-adaptativos n_a, com valor 1, e α_{bsf}, inicialmente igual a α_{UB}, e o conjunto C' com os componentes que formam a solução inicial S_{bsf}. Em seguida, o loop principal do algoritmo executa as 4 etapas principais do algoritmo como descritas na Seção 3.5.1 com algumas poucas modificações.

Além da etapa de `Construção`, que aqui utiliza α_{bsf} para controlar a geração de componentes, a etapa de `Combinação` já não inclui a atualização do vetor de idade. Isto porque na versão *self-adaptive* os componentes que não são selecionados para fazer parte da solução são descartados de C' na etapa de `Adaptação` (linha 24). Ainda, a etapa de `Solução` (linha 9) recebe o parâmetro adicional t_{ILP}, que limita o tempo de execução do procedimento $SolverPLI(C', t_{ILP})$ em t_{ILP} segundos, e retorna, além da solução S'_{opt}, o tempo t_{solve} de computação.

Nas linhas de 10 a 21 é executado o processo de auto-adaptação. Caso o solver consiga resolver o problema em tempo menor que uma certa proporção t_{prop} do tempo máximo t_{ILP}, então pode-se concluir que o conjunto C' atual pode ser resolvido facilmente e, portanto, o espaço de busca está muito pequeno e precisa ser diversificado (diminuindo α_{bsf} por um fator de α_{red}). No entanto, caso a nova solução S'_{opt} seja melhor que S_{bsf}, o número de soluções geradas n_a é redefinido para 1. Isso faz com que o algoritmo intensifique a busca na vizinhança da melhor solução recém obtida. Se, por outro lado, S'_{opt} for pior que S_{bsf}, se faz necessário aumentar a intensificação da busca ao redor de S_{bsf}. Isso pode ser feito aumentando ligeiramente α_{bsf} caso n_a já seja igual a 1, ou resetando o valor de n_a para 1 e de α_{bsf} para α_{UB}. Por fim, caso $S'_{opt} = S_{bsf}$, então o valor de n_a é incrementado com o objetivo de diversificar a busca gerando uma quantidade maior de soluções.

Algoritmo 3 Pseudocódigo do Self-adaptive CMSA

```
1:  procedure CMSA(I, t_prop, t_ILP, α_LB, α_UB, α_red)
2:      S_bsf := GeraSoluçãoInicial(C)
3:      C' := S_bsf, α_bsf := α_UB, n_a := 1
4:      while critério de parada não alcancado do
5:          for i = 1, ..., n_a do
6:              S := GeradorDeSoluções(C, S_bsf, α_bsf)              ▷ Construção
7:              C' := C' ∪ S                                         ▷ Combinação
8:          end for
9:          (S'_opt, t_solve) := SolverPLI(C', t_ILP)                ▷ Solução
10:         if t_solve < t_prop · t_ILP and α_bsf > α_LB then α_bsf := α_bsf − α_red   ▷ Auto-adaptação
11:         end if
12:         if f(S'_opt) < f(S_bsf) then
13:             S_bsf := S'_opt
14:             n_a := 1
15:         else
16:             if f(S'_opt) > f(S_bsf) then
17:                 if n_a = 1 then α_bsf := min{α_bsf + α_red/10, α_UB}
18:                 else n_a := 1
19:                 end if
20:             else
21:                 n_a := n_a + 1
22:             end if
23:         end if
24:         C' := S_bsf                                              ▷ Adaptação
25:     end while
26:     return S_bsf
27: end procedure
```

4

Metodologia

Neste capítulo é apresentado o algoritmo *Adapt*-CMSA com Tabu[1] proposto para resolver o Problema da Cadeia de Caracteres Mais Próxima. A aplicação do CMSA ao PCCP está condicionada à definição de alguns conceitos fundamentais, dentre eles: o de componente, problema reduzido e solução, introduzidos na Seção 4.1. Em seguida, é apresentada na Seção 4.2 uma heurística para construção de uma solução inicial. Na Seção 4.3 é introduzida a estrutura Tabu utilizada. E, por fim, as 4 etapas do algoritmo são destrinchadas na Seção 4.4.

4.1 Representação da Solução

Como introduzido na Seção 3.5.1, a ideia principal do *Construct, Merge, Solve & Adapt* é explorar características do problema em questão e utilizar um problema reduzido, que não necessariamente é o mesmo, para auxiliar a solução do principal. Para isso, é preciso definir de que forma o PCCP pode utilizar o problema reduzido através dos conjuntos C, C' e S.

Define-se um componente como um par ordenado $c = (s, j) \in \Sigma \times [1, m]$, onde s representa o caractere do componente e j a posição deste caractere. Por exemplo, o componente $c = (\text{‘G’}, 4)$ representa o caractere ‘G’ na posição 4. O conjunto C é definido como a união de todas as combinações possíveis de pares (s, j). Esse conjunto engloba todas as configurações de componentes que podem ser utilizadas para formar as soluções do PCCP. Uma solução válida para o PCCP consiste em um subconjunto de componentes distintos, em que, para cada posição $p \in [1, m]$, existe exatamente um componente de C ocupando essa posição. Dessa forma, o problema reduzido consiste em escolher uma seleção de componentes que cubra as m posições da string t, minimizando a maior distância de *Hamming*.

O conjunto S consiste nos componentes gerados na iteração atual e representa uma solução viável para o problema original, em que cada posição é coberta por pelo menos um componente.

[1] A implementação do algoritmo está disponível em https://github.com/Emilybtoliveira/CSP-SACMSA-Tabu

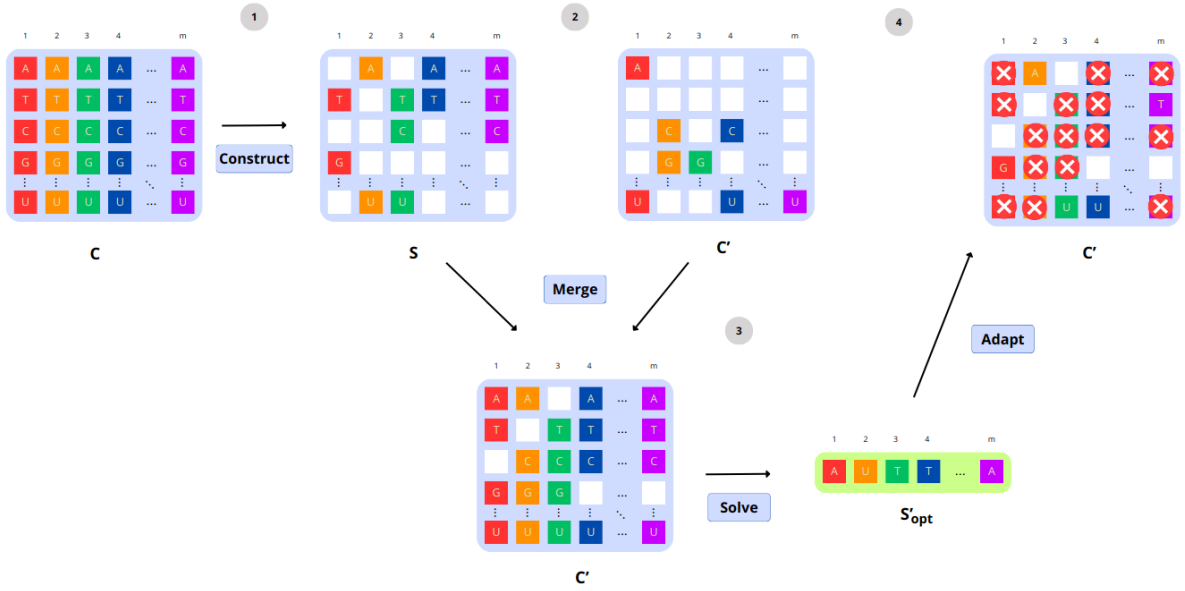


Figura 4.1: Uma iteração do CMSA. Fonte: Autor

Já o conjunto C' agrupa todos os componentes ativos de cada solução S gerada através das iterações anteriores e na iteração atual, funcionando como um *pool* de componentes candidatos à solução. Por sua vez, o conjunto S'_{opt} contém os componentes selecionados pelo algoritmo *SolverPLI*, representando a melhor solução viável até então para a instância original, embora não seja necessariamente a solução ótima. Essa solução é escolhida de forma a minimizar a maior distância de *Hamming* possível entre os candidatos presentes em C'.

A Figura 4.1 ilustra o fluxo do algoritmo em uma iteração, destacando os conjuntos C, S, C' e S'_{opt}. Durante a fase de `Construção`, um subconjunto S é gerado aleatoriamente a partir de todos os possíveis componentes representados pelo conjunto C. Em seguida, ocorre a etapa de `Combinação`, na qual o conjunto S é combinado com o conjunto C'. Posteriormente, os componentes resultantes de C' são submetidos ao *solver* na etapa de `Solução`, que encontra a melhor combinação de componentes para formar uma solução viável para o PCCP. Essa solução é representada pelo conjunto de componentes S'_{opt}. Por fim, o mecanismo de `Adaptação` é aplicado, removendo os componentes de C' que não estão presentes em S'_{opt}. Note que na versão *self-adaptive* CMSA a idade máxima não é utilizada. Além disso, é importante destacar que o conjunto C' é atualizado constantemente ao longo do algoritmo.

4.2 Construção da solução inicial

Com o objetivo de melhor explorar o espaço de busca do problema e acelerar a convergência, o *self-adaptive* CMSA inicia sua execução gerando uma solução inicial a partir do conjunto de componentes C (vide linha 2). Neste trabalho foi utilizada uma heurística construtiva gulosa, a qual constrói uma solução a partir dos caracteres mais frequentes em cada posição, uma

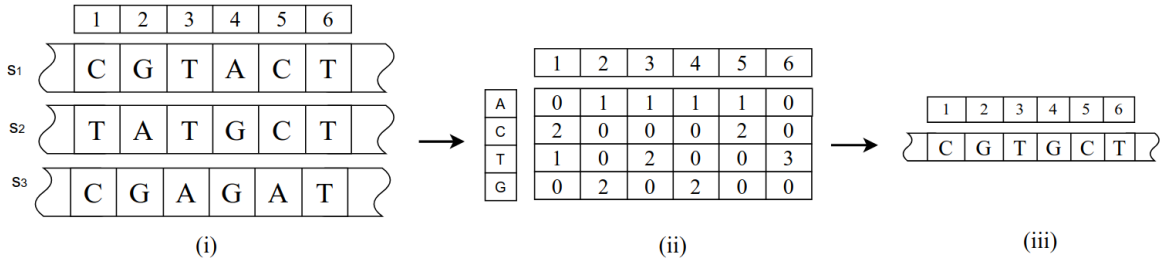


Figura 4.2: Construção de uma solução inicial a partir da matriz de frequência. Fonte: Autor

abordagem bastante explorada na literatura [7, 25, 47, 55]. Para isso, é gerada uma matriz de frequência M_F $(k \times m)$, que é preenchida percorrendo-se a instância $\mathcal{S}$ de entrada e, para cada posição j em uma *string* s_i, $M_F[s_i^j, j]$ é incrementado em uma unidade. Para fins de otimização, a matriz de frequência é gerada numa etapa de pré-processamento, anterior ao início da execução do algoritmo.

A Figura 4.2 ilustra o processo de construção da solução inicial. Inicialmente, em 4.2 (i), tem-se a instância $\mathcal{S}$, com 3 *strings* de comprimento 6, formadas sob um alfabeto de tamanho 4 (A, C, T, G). Em seguida, em 4.2 (ii), inicia-se a construção da matriz de frequência M_F, com dimensão 4×6. Cada entrada (i, j) na matriz representa a quantidade de strings que possuem o caractere Σ_i na posição j. Por exemplo, na primeira posição das *strings* em $\mathcal{S}$, os caracteres "A" e "G" não têm nenhuma ocorrência, enquanto que "C" e "T" possuem 2 ($\mathcal{S}_1^1$ e $\mathcal{S}_3^1$) e 1 ($\mathcal{S}_2^1$) ocorrências, respectivamente. Por fim, em 4.2 (iii), no início da execução do *self-adaptive* CMSA a matriz de frequência é utilizada para construir uma solução inicial, escolhendo os caracteres com maior frequência em cada coluna.

O Algoritmo 4 apresenta o pseudocódigo do procedimento *GeraSoluçãoInicial*, o qual recebe a matriz de frequência M_F e retorna uma solução inicial S'. Inicialmente, nas linhas 2 e 3, o algoritmo inicializa o vetor de caracteres mais frequentes V_F e o conjunto de componentes S'. A construção do vetor V_F é de extrema importância na etapa de `Construção` descrita posteriormente na Seção 4.4.1. Em seguida, para cada posição $j \in [1, \ldots, m]$, a linha 5 retorna o índice i que possui a maior contagem de ocorrências na coluna j da matriz de frequência. Em caso de empate entre caracteres, o procedimento escolhe um aleatoriamente. Na linha 6, busca-se o caractere do alfabeto correspondente ao índice i e na linha 7, este caractere é salvo na posição j do vetor V_F. Em seguida, o mesmo caractere é utilizado para compor um novo componente de S' (linha 8). O procedimento conclui na linha 10 retornando S'.

4.3 Busca tabu

Na Seção 3.4 foi introduzida a meta-heurística Tabu *Search* que consiste em utilizar uma memória de curto prazo para guiar a exploração do espaço de busca. A estrutura lista tabu foi

Algoritmo 4 Procedimento gerador de uma solução inicial

```
1:  procedure GERASOLUÇÃOINICIAL(M_F)
2:      V_F := ∅
3:      S' := ∅
4:      for j = 1,...,m do
5:          i := arg max_{i∈[1,...,k]}(M_F[i, j])
6:          caractere := Σ_i
7:          V_F[j] := caractere
8:          S' := S' ∪ (caractere, j)
9:      end for
10:     return S'
11: end procedure
```

incorporada ao *self-adaptive* CMSA como estratégia de diversificação.

Uma lista tabu é utilizada para evitar que componentes desprezados como candidatos à solução fiquem re-entrando no *pool* de soluções repetidamente por conta da estocasticidade do procedimento *GeradorDeSoluções*. Assim, o algoritmo é forçado a escolher outros componentes, explorando diferentes vizinhanças.

No algoritmo proposto, a lista tabu é implementada como uma matriz M_T $(k \times m)$, em que o valor $M_T[i, j]$ representa a iteração em que o caractere Σ_i entrou na lista tabu da posição j. Inicialmente, toda a matriz é inicializada com o valor $-\infty$. Em uma dada iteração, se uma entrada (i, j) da matriz tabu contiver o valor $-\infty$, então um componente (Σ_i, j) pode ser gerado imediatamente. Caso contrário, conclui-se que o caractere entrou na lista tabu em algum momento ao longo da execução do algoritmo e se faz necessário verificar há quantos *loops* isso ocorreu. Para isso, é definido um limiar tt que determina o tempo exato que um componente deve ser "bloqueado". Caso a diferença entre o loop atual l e $M_T[i, j]$ seja menor que tt, o componente ainda está na lista tabu e não pode compor uma solução.

A escolha do limiar é realizada através de uma heurística simples que se autoajusta para cada instância. Na etapa de pré-processamento, anterior à execução do CMSA, varre-se o conjunto de *strings* de entrada $\mathcal{S}$ verificando, para cada posição $j \in [1, \ldots, m]$, a quantidade de caracteres distintos que aparecem na *string* s_i. Salva-se o menor valor encontrado e este torna-se o limiar tt da lista tabu. Dessa forma, o efeito prático dessa heurística é impedir que posições com poucas escolhas de caracteres fiquem por várias iterações sem gerar componentes.

4.4 Adapt-CMSA com Tabu

As subseções seguintes descrevem detalhamente as etapas de construção, combinação, solução e adaptação do *self-adaptive* CMSA, descrito na Seção 3.5.1, utilizando a representação da solução, solução inicial e lista tabu apresentadas nas seções anteriores.

4.4.1 Construct

A fase de `Construção` é a etapa inicial do algoritmo, que ocorre quando o procedimento *GeradorDeSoluções*(C, S_{bsf}, α_{bsf}) é chamado. O Algoritmo 5 apresenta o pseudocódigo do procedimento, cujo objetivo é gerar componentes a partir de C.

Algoritmo 5 Procedimento gerador de soluções

```
 1: procedure GeradorDeSolucoes(C, S_bsf, α_bsf)
 2:     S' := ∅
 3:     caractere := null
 4:     for j ∈ [1,m] do
 5:         r := rand(0,1)
 6:         if r > α_bsf then
 7:             if not isCharMaisFrequenteVisitado(j) then
 8:                 caractere := getCharMaisFrequente(j)
 9:             else
10:                 caractere := getCharAleatorio(i)
11:             end if
12:         else
13:             caractere := S_bsf[j]
14:         end if
15:         c := (caractere, j)
16:         S' := S' ∪ c
17:     end for
18:     return S'
19: end procedure
```

O algoritmo inicia criando o conjunto auxiliar S' que irá armazenar os componentes gerados e a variável *caractere* que conterá o caractere do componente. Para cada posição j no intervalo $[1,m]$, sorteia-se aleatoriamente um número $r \in \mathbb{R}$ no intervalo $[0,1]$, o qual determinará a geração de um componente para a posição j. Isso acontece se o valor de r ultrapassar o parâmetro auto-adaptativo α_{bsf}. Dessa forma, quão maior for o valor de α_{bsf}, menos componentes são gerados, e a solução S' é mais parecida com S_{bsf}.

Em seguida, o procedimento checa se o caractere mais frequente da coluna j já está presente no conjunto C' através do procedimento auxiliar `isCharMaisFrequenteVisitado(j)`. Este recebe a posição j e busca no vetor V_F, apresentado na Seção 4.2, o caractere correspondente, verificando, em seguida, sua presença em C'. Essa verificação é feita de forma eficiente através da manutenção de uma matriz M_V $(k \times m)$ de componentes visitados. Através dela é possível checar em tempo constante $O(1)$ se o componente gerado está presente em C' ou não.

Se o caractere mais frequente estiver disponível, este é escolhido para compor o componente. Caso contrário, gera-se um caractere aleatório através de outro procedimento auxiliar, `getCharAleatorio(j)`, que seleciona aleatoriamente um caractere disponível. Um caractere disponível, nesse caso, é um que não esteja em C' e nem na lista tabu da posição j. Vale destacar ainda que, para cada posição j, caracteres aleatórios são gerados a partir de uma estrutura de dados que armazena os caracteres do alfabeto que aparecem na posição j das *strings* de $\mathcal{S}$. Sua utilização otimiza o desempenho do algoritmo, visto que impede o aumento da distância de

Hamming em 1 unidade quando um componente não possui ocorrências em uma determinada posição. Por fim, o procedimento conclui retornando o conjunto S' contendo os componentes gerados.

4.4.2 Merge

A etapa de `Combinação` ocorre imediatamente após a fase de `Construção`, conforme indicado na linha 7 do Algoritmo 3. Nessa etapa, os novos componentes gerados pelo procedimento *GeradorDeSoluções*(C, S_{bsf}, α_{bsf}) são combinados com os componentes gerados nas iterações anteriores.

Embutido nesse procedimento está o cálculo da distância de *Hamming* de cada componente (s, j). As disparidades entre s e a posição j de cada string de entrada são armazenadas e utilizadas posteriormente na etapa de `Solução`.

4.4.3 Solve

A etapa de `Solução` é executada na chamada para o procedimento *SolverPLI*(C', t_{ILP}), na linha 9. Este recebe o conjunto C' e o tempo máximo de execução t_{ILP} e retorna uma seleção de componentes S'_{opt} que corresponda a uma solução para a instância do PCCP e o tempo t_{solve} levado para resolver a instância do problema reduzido.

O *solver* utilizado resolve uma formulação do Problema de Particionamento de Conjuntos (PPC) [20]. Algumas matheurísticas acopladas a um *solver* solucionando uma formulação do PPC foram propostas na literatura e aplicadas a diferentes problemas com sucesso [15, 49, 59].

Para resolver o problema reduzido do PCCP, utiliza-se a seguinte formulação inteira do PPC:

$$\min \quad z \tag{4.1}$$

$$\text{s.a} \quad \sum_{c \in C_j} x_c = 1 \qquad \forall j \in \{1, \ldots, m\} \tag{4.2}$$

$$\sum_{c \in C'} d_c^i x_c \leq z \qquad \forall i \in \{1, \ldots, n\} \tag{4.3}$$

$$x_c \in \{0, 1\} \qquad \forall c \in C' \tag{4.4}$$

$$z \in \mathbb{Z}_+ \tag{4.5}$$

Para cada componente $c = (s, j) \in C'$, é introduzida a variável de decisão binária x_c, que indica se o componente c foi selecionado para fazer parte da solução final. Além disso, é utilizada a constante d_c^i, que representa a distância de Hamming entre o componente c e a *string* de entrada s_i calculada na etapa de `Combinação`. Por fim, a notação C_j refere-se aos componentes de C' que contêm a posição j, e a variável z representa a maior distância de *Hamming* a ser minimizada.

A função objetivo (4.1) minimiza o valor de z. As restrições (4.2) asseguram que, para

cada posição, exatamente um componente deve ser selecionado. As restrições (4.3) impõem um limite superior para a soma das distâncias de cada *string* da instância do PCCP, garantindo que esse valor não ultrapasse z. Isso significa que, quando $x_c = 0$, as distâncias relacionadas ao conjunto c não são consideradas no cálculo. Por fim, as restrições (4.4) e (4.5) são restrições de domínio.

Vale ressaltar que na versão *self-adaptive* CMSA, o algoritmo exato deve ser executado por, no máximo, t_{ILP} segundos. Para garantir isso e evitar que o *solver* utilize boa parte desse tempo tentando melhorar uma solução que já é boa, o $SolverPLI(C', t_{ILP})$ foi configurado para interromper sua execução antes de t_{ILP} se já tiver encontrado uma solução melhor que S_{bsf}.

4.4.4 Adapt

Na versão original do CMSA, apresentada na Seção 3.5, a etapa de `Adaptação` desempenha um papel fundamental no algoritmo, pois envolve o mecanismo de envelhecimento dos componentes. Os componentes selecionados pelo algoritmo exato para fazer parte da solução têm suas idades redefinidas para zero, enquanto os demais têm suas idades incrementadas em uma unidade. Além disso, os componentes que ultrapassam uma idade máxima são removidos do conjunto de candidatos.

No entanto, na versão *self-adaptive* do CMSA, a idade máxima dos componentes é fixa em 1, isto é, aqueles que não são selecionados para fazerem parte da solução são descartados imediatamente. Nesse caso, a etapa de adaptação consiste em simplesmente remover esses componentes do conjunto C'.

Além disso, especificamente no *Adapt*-CMSA com Tabu, a adaptação tem como última etapa a atualização das matrizes de componentes *visitados* M_V e *tabu* M_T. Se um componente $c = (s, j)$ é removido, a posição $M_V[s, j]$ é atualizada para 0 e a mesma posição de M_T recebe o índice da iteração atual, indicando que c está na lista tabu.

5

Resultados e Discussões

Neste capítulo são apresentados os resultados computacionais obtidos a partir da implementação do algoritmo *Adapt*-CMSA com Tabu, descrito no Capítulo 4. A Seção 5.1 descreve as instâncias-teste utilizadas nos experimentos. A Seção 5.2 aponta o ambiente onde os experimentos foram realizados e as ferramentas utilizadas. A Seção 5.3 descreve o processo de ajuste dos parâmetros do algoritmo. E, por fim, a Seção 5.4 apresenta os resultados computacionais obtidos e a discussão.

5.1 Instâncias

As instâncias utilizadas nos experimentos foram divididas em dois grupos: artificiais e biológicas. O grupo de instâncias artificiais foi retirado do trabalho de Chimani et al. [11], as quais foram disponibilizadas pelos autores[1] e utilizadas em outros trabalhos da literatura [55, 60]. O *dataset* de Chimani consiste de 5250 instâncias geradas aleatoriamente no alfabeto $\Sigma \in \{2,4,20\}$, com $m \in \{10,20,30,40,50\}$ e $n \in \{250,500,750,1000,2000,5000,10000\}$. As diferentes dimensões de alfabeto simulam os tipos de dados encontrados nas aplicações reais do PCCP. As instâncias binárias, por exemplo, visam simular os códigos binários encontrado na Teoria dos Códigos, enquanto as de alfabeto 4 e 20 simulam sequências de DNA/RNA e proteínas, respectivamente. Para cada configuração, composta pela combinação dos valores de Σ, m e n, os autores geraram 50 instâncias distintas. Para os experimentos deste trabalho foi selecionada apenas a primeira instância de cada configuração (instâncias cujo *ID* é "1-0"), resultando em um total de 105 instâncias distintas.

O segundo grupo, abrangendo as instâncias biológicas, é denominado *dataset* de Mcclure. O grupo é constituído de 6 instâncias, representando diferentes sequências de proteínas reais ($|\Sigma| = 20$). Retirado do trabalho de Mcclure et al. [39], o conjunto tem sido amplamente utilizado na literatura de problemas de *string consensus* [11, 23, 40, 41, 47].

[1] https://tcs.uos.de/research/csp_cssp

5.2 Ambiente e ferramentas

Todos os testes foram executados em um computador equipado com um processador Intel Core i5-10300H 2.50GHz, 8GB de RAM e sistema operacional Ubuntu 20.04.6 LTS. Os algoritmos foram implementados na linguagem C++ e o compilados com o GNU G++ versão 9.4.0, utilizando a flag de otimização '-O3'. Para a resolução das formulações matemáticas foi utilizado o otimizador IBM ILOG CPLEX versão 12.9.

Para que o *solver* da etapa de `Solução` pudesse ser interrompido assim que encontrasse uma boa solução (vide Seção 4.4.3), foi utilizado o mecanismo de *callback* do CPLEX. O *callback* utilizado é do tipo `MIPInfoCalback`[2], o qual é chamado regularmente durante o processo de *branch-and-bound* e cuja implementação é definida pelo usuário. O `MIPInfoCallback` foi implementado de modo a verificar se o tempo que o algoritmo consumiu até então atinge o tempo máximo ou se o valor da função objetivo da solução incumbente é melhor que S_{bsf}. Se pelo menos uma dessas condições for satisfeita, a execução do CPLEX é abortada e a solução incumbente é retornada.

5.3 Parâmetros

Os parâmetros do *Adapt*-CMSA com Tabu foram calibrados através da ferramenta *irace*[3]. O *irace* (López-Ibáñez et al. [37]) é um pacote *open-source* implementado em R que tem o objetivo de configurar automaticamente hiperparâmetros de algoritmos de otimização. Funciona recebendo um grupo de instâncias e de parâmetros com seus respectivos domínios e utilizando, em seguida, ferramentas estatísticas para escolher uma configuração de parâmetros adequada.

Para uma calibragem mais rápida e não enviesada do *Adapt*-CMSA com Tabu foram selecionadas aleatoriamente 8 instâncias a partir do grupo de instâncias-teste. Os parâmetros calibrados, seus domínios e a melhor configuração escolhida pelo *irace* são especificados na Tabela 5.1.

Tabela 5.1: Parâmetros calibrados pelo *irace*

Parâmetro	Domínio	Valor escolhido
t_{ILP}	(0.05, 3)	**3.0**
t_{prop}	(0, 1)	**0.5596**
α_{LB}	(0, 1)	**0.2950**
α_{UB}	(0, 1)	**0.9872**
α_{red}	(0, 1)	**0.9385**

[2] Documentação do MIPInfoCallback
[3] https://mlopez-ibanez.github.io/irace/

5.4 Resultados experimentais

Com o objetivo de avaliar seu desempenho, os resultados do *Adapt*-CMSA com Tabu proposto neste trabalho foram comparados com os algoritmos meta-heurísticos mais recentes encontrados na literatura: o *Simulated Annealing with Greedy Walk* (SAGW) proposto por Pappalardo et al. [47] e o *Hybrid Discrete Particle Swarm Optimization* (HDPSO) proposto por Santos [55]. O código-fonte dos algoritmos não pôde ser obtido e, portanto, foram implementados de acordo com suas descrições nos respectivos trabalhos. A configuração dos parâmetros dos algoritmos implementados é a mesma que a utilizada pelos autores de cada um, com exceção do critério de parada. Para fins de simplificação e comparação justa, o tempo de execução dos 3 algoritmos foi limitado a 30 segundos.

As Tabelas 5.2 a 5.5 apresentam os resultados obtidos para os grupos de instâncias-teste. As colunas n e m indicam a quantidade e o tamanho das *strings*, respectivamente. Já a coluna nomeada "*CPLEX*" indica o valor da solução exata da instância. Os valores exatos são retirados, em sua maioria, do trabalho de Chimani et al. [11]. No entanto, os autores limitaram o tempo de execução do algoritmo exato em 30 minutos e, por essa razão, algumas soluções não são ótimas. Assim, as instâncias enquadradas nesse cenário foram re-executadas no CPLEX, rodando a formulação para o PCCP apresentada na Seção 2.2, com tempo limite de 1 hora. Porém, o exato ainda não foi capaz de encontrar o ótimo de algumas instâncias e, portanto, estas possuem "*" na coluna "*CPLEX*" e o respectivo valor corresponde ao limite inferior retornado pelo *solver*.

As colunas agrupadas pelo nome da meta-heurística ("HDPSO", "SAGW" e "SA-CMSA TABU") mostram a melhor solução encontrada em 10 execuções ("BEST"), o valor médio das soluções ("AVG.") e o gap entre o valor exato e a heurística ("GAP"). O gap em relação à solução exata é calculado da seguinte forma:

$$gap = \frac{x - x^*}{x^*} \tag{5.1}$$

em que: x é o valor encontrado pela meta-heurística para o problema e x^* é o valor da solução ótima ou do limite inferior quando o ótimo não é conhecido. Nas tabelas, os melhores gaps são destacados em negrito.

Na Tabela 5.2 são apresentados os resultados para as instâncias artificiais binárias de Chimani et al. [11]. Apesar de parecerem as instâncias mais fáceis devido ao alfabeto reduzido, as binárias são o grupo que os algoritmos demonstram mais dificuldade em resolver. Para 14 dessas instâncias o CPLEX não encontra o valor ótimo no tempo limite estipulado. Apesar disso, o SA-CMSA TABU performa melhor que as abordagens SAGW, a qual não encontra nenhum valor ótimo, e HDPSO, que o encontra em uma única instância. O SA-CMSA TABU, por sua vez, encontra o valor garantidamente ótimo em cerca de 48% das instâncias. Ainda, fica a 1 unidade do valor encontrado pelo exato em 10 das 18 em que não encontra o ótimo, isto é, encontra a solução que corresponde ao limite superior do CPLEX em 55% das instâncias. Além disso, o

algoritmo proposto se sobressai também na média de resultados. Em todas as instâncias possui valores médios menores que os outros dois algoritmos, e em 5 das 17 em que encontra o ótimo o desvio padrão é 0, o que significa que a solução ótima é encontrada em todas as 10 execuções.

Os resultados para as instâncias de DNA são indicados na Tabela 5.3, os quais demonstram um avanço considerável em relação às binárias. O SA-CMSA TABU encontra o valor ótimo em 20 das 35 (cerca de 57%), ficando a apenas uma unidade do valor exato em todo o restante. Isso resulta em uma média de gap consideravelmente reduzida em relação aos outros dois algoritmos, sendo 91% menor que a do SAGW e 99% que a do HDPSO. Ainda, o algoritmo proposto tem desvio padrão igual a 0 em 10 das instâncias em que alcança o ótimo.

Por fim, a Tabela 5.4 apresenta os resultados para as instâncias de proteínas. Para este grupo, os algoritmos apresentam o melhor desempenho, de modo que as suas médias de gaps são as menores dentre os 3 grupos de instâncias. No entanto, a interpretação dos resultados é bem semelhante, de modo que o método proposto neste trabalho supera o desempenho das outras duas abordagens. O SA-CMSA TABU encontra o valor exato em 31 das 35 instâncias ($\approx 88\%$) e encontra soluções 1 unidade distante do valor exato nas outras 4. Dessa forma, o gap médio do método proposto é 95% e 99% menor que os das abordagens SAGW e HDPSO, respectivamente. Além disso, o desvio padrão é 0 para a grande maioria das instâncias em que o SA-CMSA TABU encontra o ótimo (24 das 31).

Sumarizando, em comparação com os algoritmos SAGW e HDPSO, o *Adapt*-CMSA com Tabu encontra soluções melhores ou iguais em todas as instâncias. O SAGW apresenta resultados próximos aos do CMSA, no entanto se distaciam à medida em que crescem as dimensões das instâncias. O HDPSO é o que apresenta o pior desempenho, obtendo as maiores médias de gaps e encontrando apenas 1 solução ótima nas 105 instâncias. Em resumo, o SA-CMSA TABU alcança o ótimo garantido em 68 das 105 instâncias artificiais ($\approx 64\%$) e encontra uma solução 1 unidade distante do valor exato em 29 ($\approx 78\%$ das que ele não encontra o ótimo).

A Tabela 5.5 apresenta os resultados para as instâncias biológicas de Mcclure. Neste experimento, o SA-CMSA TABU também obtém melhores resultados em comparação com os outros dois algoritmos, encontrando o ótimo em 4 das 6 instâncias ($\approx 66\%$) e ficando a 1 unidade do valor exato nas outras duas. Além disso, o valor médio dos resultados do CMSA é o menor dentre os 3 algoritmos. Em 3 instâncias, o desvio padrão é 0, indicando que o algoritmo encontra o valor ótimo em todas as execuções. Ainda, observa-se uma mudança no desempenho dos algoritmos SAGW e HDPSO. Enquanto nas instâncias do experimento anterior o SAGW superava com folga os resultados do HDPSO, aqui os algoritmos possuem resultados bastante parecidos. Nas instâncias do grupo "Mcclure-586-20" o SAGW ainda é mais bem sucedido e encontra o ótimo em duas delas. No entanto, no grupo "Mcclure-582-20", o HDPSO encontra resultados melhores que o SAGW e bem próximos ao do SA-CMSA TABU, o que resulta numa média de gap menor que a do SAGW.

Tabela 5.2: Resultados para as instâncias de Chimani et al. com $|\Sigma| = 2$

		CPLEX	**HDPSO**			**SAGW**			**SA-CMSA TABU**		
n	m	BEST	BEST	AVG.	GAP	BEST	AVG.	GAP	BEST	AVG.	GAP
10	250	96	96	97.2	**0**	97	97.9	0.0104	96	96.0	**0**
10	500	191	195	195.9	0.0209	192	192.8	0.0052	191	191.1	**0**
10	750	284	293	294.9	0.0317	285	285.1	0.0035	284	284.0	**0**
10	1000	378	397	400.9	0.0503	379	379.7	0.0026	378	378.5	**0**
10	2000	758	841	843.7	0.1095	760	760.4	0.0026	758	758.2	**0**
10	5000	1891	2184	2186.4	0.1549	1895	1896.6	0.0021	1891	1891.7	**0**
10	10000	3775	4436	4439.8	0.1751	3788	3792.7	0.0034	3775	3776.4	**0**
20	250	105	108	109.3	0.0286	107	107.9	0.0190	105	106.0	**0**
20	500	206	218	219.6	0.0583	210	210.3	0.0194	206	206.3	**0**
20	750	308	333	335.6	0.0812	313	314.1	0.0162	308	308.1	**0**
20	1000	416	454	457.1	0.0913	421	421.9	0.0120	416	416.2	**0**
20	2000	826	926	927.9	0.1211	834	834.6	0.0097	826	826.9	**0**
20	5000	2065	2350	2354.2	0.1380	2076	2079.2	0.0053	2066	2068.8	**0.0005**
20	10000	4116	4723	4726.8	0.1475	4133	4135.7	0.0041	4116	4117.8	**0**
30	250	110	114	115.3	0.0364	114	114.0	0.0364	110	110.0	**0**
30	500	218	234	235.4	0.0734	221	221.8	0.0138	218	219.0	**0**
30	750	322	352	352.8	0.0932	327	327.3	0.0155	322	322.9	**0**
30	1000	426*	470	471.6	0.1033	434	434.4	0.0188	427	427.8	**0.0023**
30	2000	856*	955	956.9	0.1157	868	869.3	0.0140	857	857.8	**0.0012**
30	5000	2139	2408	2411.6	0.1258	2171	2173.7	0.0150	2140	2141.8	**0.0005**
30	10000	4269*	4825	4831.5	0.1302	4297	4300.7	0.0066	4271	4271.8	**0.0005**
40	250	113	119	120.6	0.0439	121	121.0	0.0614	114	114.0	**0.0088**
40	500	222*	238	240.4	0.0721	230	230.4	0.0360	223	223.2	**0.0045**
40	750	331	361	362.0	0.0906	344	344.7	0.0393	332	332.2	**0.0030**
40	1000	437*	478	480.8	0.0938	446	446.9	0.0206	438	438.7	**0.0023**
40	2000	879*	972	972.9	0.1058	896	898.0	0.0193	881	882.0	**0.0023**
40	5000	2184*	2439	2440.5	0.1168	2217	2219.4	0.0151	2186	2187.3	**0.0009**
40	10000	4368*	4883	4888.5	0.1179	4406	4408.7	0.0087	4370	4370.8	**0.0005**
50	250	114	119	121.1	0.0531	121	121.0	0.0708	114	114.8	**0**
50	500	225*	242	244.6	0.0756	234	234.6	0.0400	226	227.0	**0.0044**
50	750	336*	366	367.3	0.0893	346	346.8	0.0298	337	337.6	**0.0030**
50	1000	447*	487	489.8	0.0895	463	463.2	0.0358	449	449.5	**0.0045**
50	2000	890*	980	983.5	0.1011	918	919.0	0.0315	893	893.0	**0.0034**
50	5000	2223*	2461	2463.1	0.1071	2264	2266.3	0.0184	2227	2227.7	**0.0018**
50	10000	4431*	4916	4923.3	0.1095	4506	4507.7	0.0169	4433	4435.5	**0.0005**
Média					0.0932			0.0155			**0.0005**
Ótimos					1			0			**17**
Melhor AVG.					0			0			**35**

Tabela 5.3: Resultados para as instâncias de Chimani et al. com $|\Sigma| = 4$

		CPLEX	**HDPSO**			**SAGW**			**SA-CMSA TABU**		
n	m	BEST	BEST	AVG.	GAP	BEST	AVG.	GAP	BEST	AVG.	GAP
10	250	144	148	149.0	0.0278	144	144.9	**0**	144	144.2	**0**
10	500	287	304	305.2	0.0592	288	288.2	0.0035	287	287.0	**0**
10	750	437	474	479.1	0.0847	438	438.5	0.0023	437	437.0	**0**
10	1000	579	637	640.0	0.1002	580	580.1	0.0017	579	579.0	**0**
10	2000	1153	1305	1309.1	0.1318	1160	1162.9	0.0061	1153	1153.0	**0**
10	5000	2910	3340	3343.7	0,1478	2914	2915.3	0,0014	2910	2910.0	**0**
10	10000	5791	6697	6702.1	0.1564	5795	5798.02	0.0007	5791	5791.0	**0**
20	250	158	168	169.3	0.0633	160	160.9	0.0127	158	158.2	**0**
20	500	315	344	345.8	0.0921	319	319.7	0.0127	315	315.5	**0**
20	750	476	523	525.1	0.0987	482	482.4	0.0126	476	476.2	**0**
20	1000	633	697	701.1	0.1011	644	644.8	0.0174	633	633.2	**0**
20	2000	1264	1409	1412.5	0.1147	1276	1276.7	0.0095	1264	1264.0	**0**
20	5000	3162	3552	3555.0	0.1233	3191	3192.2	0.0092	3162	3162.4	**0**
20	10000	6310	7118	7121.9	0.1281	6338	6343.8	0.0044	6310	6310.0	**0**
30	250	165	177	177.6	0.0727	169	169.5	0.0242	165	165.3	**0**
30	500	326	354	356.6	0.0859	334	334.8	0.0245	327	327.0	**0.0031**
30	750	492	538	539.6	0.0935	501	501.6	0.0183	493	493.0	**0.0020**
30	1000	654	719	721.0	0.0994	659	659.7	0.0076	654	654.7	**0**
30	2000	1308	1442	1446.5	0.1024	1321	1322.4	0.0099	1308	1308.1	**0**
30	5000	3268	3627	3629.6	0.1099	3292	3294.2	0.0073	3268	3268.0	**0**
30	10000	6541	7259	7262.7	0.1098	6582	6585.5	0.0063	6541	6541.0	**0**
40	250	168*	181	181.7	0.0774	176	176.4	0.0476	169	169.7	**0.0060**
40	500	333	361	363.1	0.0841	342	342.2	0.0270	334	334.0	**0.0030**
40	750	499*	545	546.8	0.0922	513	514.0	0.0281	500	500.8	**0.0020**
40	1000	668	730	730.5	0.0928	681	681.6	0.0195	669	669.0	**0.0015**
40	2000	1334*	1462	1464.5	0.0960	1357	1357.9	0.0172	1335	1335.9	**0.0007**
40	5000	3334	3663	3666.9	0.0987	3364	3366.0	0.0090	3334	3334.9	**0**
40	10000	6661*	7328	7333.5	0.1001	6703	6707.8	0.0063	6662	6662.0	**0.0002**
50	250	172	185	185.6	0.0756	180	180.0	0.0465	173	173.1	**0.0058**
50	500	339*	367	368.5	0.0826	351	351.0	0.0354	340	340.6	**0.0029**
50	750	508	552	552.9	0.0866	516	516.9	0.0157	509	509.1	**0.0020**
50	1000	677*	736	737.5	0.0871	693	693.8	0.0236	678	678.4	**0.0015**
50	2000	1351*	1473	1475.7	0.0903	1372	1373.7	0.0155	1352	1352.0	**0.0007**
50	5000	3377*	3687	3689.8	0.0918	3407	3409.0	0.0089	3378	3378.0	**0.0003**
50	10000	6752*	7376	7378.7	0.0924	6803	6806.0	0.0076	6753	6753.0	**0.0001**
Média					0.0928			0.0099			**0.0008**
Ótimos					0			1			**20**
Melhor AVG.					0			0			**35**

Tabela 5.4: Resultados para as instâncias de Chimani et al. com $|\Sigma| = 20$

		CPLEX	**HDPSO**			**SAGW**			**SA-CMSA TABU**		
n	m	BEST	BEST	AVG.	GAP	BEST	AVG.	GAP	BEST	AVG.	GAP
10	250	195	208	208.5	0.0667	195	195.6	**0**	195	195.0	**0**
10	500	390	421	421.9	0.0795	390	390.9	**0**	390	390.0	**0**
10	750	587	633	634.6	0.0784	588	588.4	0.0017	587	587.0	**0**
10	1000	787	847	850.4	0.0762	787	787.9	**0**	787	787.0	**0**
10	2000	1563	1701	1702.2	0.0883	1564	1565.0	0.0006	1563	1563.0	**0**
10	5000	3915	4268	4271.0	0.0902	3918	3918.6	0.0008	3915	3915.0	**0**
10	10000	7819	8541	8543.4	0.0923	7823	7825.2	0.0005	7819	7819.0	**0**
20	250	210	224	224.4	0.0667	213	213.8	0.0143	210	210.0	**0**
20	500	420	448	449.5	0.0667	421	421.9	0.0024	420	420.0	**0**
20	750	630	676	676.2	0.0730	633	633.8	0.0048	630	630.0	**0**
20	1000	839	900	901.6	0.0727	843	843.9	0.0048	839	839.0	**0**
20	2000	1678	1802	1805.3	0.0739	1683	1683.8	0.0030	1678	1678.0	**0**
20	5000	4194	4515	4517.2	0.0765	4204	4205.1	0.0024	4194	4194.0	**0**
20	10000	8389	9032	9034.7	0.0766	8405	8409.7	0.0019	8389	8389.0	**0**
30	250	215	229	229.5	0.0651	218	218.9	0.0140	215	215.7	**0**
30	500	430	459	459.8	0.0674	433	434.3	0.0070	430	430.1	**0**
30	750	647	690	690.4	0.0665	652	652.9	0.0077	647	647.0	**0**
30	1000	861	918	919.9	0.0662	869	869.4	0.0093	861	861.0	**0**
30	2000	1725	1840	1841.5	0.0667	1735	1735.3	0.0058	1725	1725.0	**0**
30	5000	4309	4599	4600.2	0.0673	4321	4323.3	0.0028	4309	4309.0	**0**
30	10000	8619	9196	9200.9	0.0669	8641	8642.9	0.0026	8619	8619.0	**0**
40	250	220	232	233.0	0.0545	223	223.2	0.0136	220	220.0	**0**
40	500	437	465	465.4	0.0641	442	442.0	0.0114	438	438.0	**0.0023**
40	750	657	697	697.6	0.0609	663	663.3	0.0091	657	657.0	**0**
40	1000	875	929	930.3	0.0617	883	883.9	0.0091	875	875.6	**0**
40	2000	1749	1858	1858.6	0.0623	1760	1761.0	0.0063	1749	1749.4	**0**
40	5000	4371	4641	4643.5	0.0618	4387	4389.5	0.0037	4371	4371.0	**0**
40	10000	8747	9281	9282.9	0.0610	8768	8771.5	0.0024	8747	8747.0	**0**
50	250	221*	234	234.9	0.0588	225	225.3	0.0181	222	222.0	**0.0045**
50	750	442	468	468.6	0.0588	447	447.5	0.0113	442	442.0	**0**
50	500	662*	701	702.3	0.0589	669	669.8	0.0106	663	663.0	**0.0015**
50	1000	883	934	935.4	0.0578	891	892.6	0.0091	883	883.9	**0**
50	2000	1767	1868	1869.6	0.0572	1781	1781.2	0.0079	1767	1767.9	**0**
50	5000	4415	4666	4669.2	0.0569	4437	4438.07	0.0050	4416	4416.3	**0.0002**
50	10000	8831	9331	9333.2	0.0566	8855	8857.5	0.0027	8831	8831.2	**0**
Média					0.0667			0.0049			**0.0002**
Ótimos					0			3			**31**
Melhor AVG.					0			0			**35**

Tabela 5.5: Resultados para as instâncias de Mcclure et al.

			CPLEX	HDPSO			SAGW			SA-CMSA TABU		
Grupo	n	m	BEST	BEST	AVG.	GAP	BEST	AVG.	GAP	BEST	AVG.	GAP
Mcclure-582-20	6	141	88	89	89.1	0.0114	98	98.0	0.1136	88	88.0	**0**
	10	141	97	99	99.8	0.0206	112	113.5	0.1546	98	98.4	**0.01**
	12	141	97	99	99.9	0.0206	110	110.0	0.1340	98	98.2	**0.01**
Mcclure-586-20	6	100	72	73	73.2	0.0139	72	72.0	**0**	72	72.0	**0**
	10	98	75	78	79.2	0.0400	76	76.0	0.0133	75	75.1	**0**
	12	98	77	80	80.8	0.0390	77	77.1	**0**	77	77.0	**0**
Média						0.0206			0.0635			**0.003**
Ótimos						0			2			**4**
Melhor AVG.						0			0			**5**

5.4.1 Impacto dos componentes

O último experimento realizado visou a análise do impacto dos componentes do *Adapt*-CMSA com Tabu no seu bom desempenho. Para isso, foram geradas 4 versões distintas do algoritmo, cada uma sendo uma versão incremental da anterior. A Tabela 5.6 apresenta um resumo dos componenentes presentes nas 4 versões, nomeadas `CMSA`, `CMSA-1`, `CMSA-2`, `CMSA-3`. A versão mais básica é a `CMSA`, sendo a implementação pura do algoritmo descrito na Seção 3.5.1 e elaborado por Oliveira et al. [44]. A versão `CMSA-1` se refere à implementação do CMSA utilizando a estrutura tabu (Seção 4.3) e a `CMSA-2` é uma versão melhorada da `CMSA-1` com a adição da solução inicial (Seção 4.2). Por fim, `CMSA-3` representa o *Adapt*-CMSA com Tabu, versão final do algoritmo, incrementando o `CMSA-2` com o *callback* de tempo no CPLEX (Seção 5.2) e a auto-adaptação dos parâmetros (Seção 3.5.1).

Tabela 5.6: Componentes presentes nas diferentes versões do CMSA

Versão	**Componentes**			
	Tabu	**Solução inicial**	**Callback de tempo**	**Auto-adaptação**
CMSA				
CMSA-1	✓			
CMSA-2	✓	✓		
CMSA-3	✓	✓	✓	✓

Para avaliar as versões do algoritmo foram gerados gráficos de distribuição de probabilidade acumulada. *Time-to-target plots* (ou *ttt-plots*) são gráficos bastante utilizados na literatura por serem úteis na caracterização do tempo de execução e na comparação de diferentes algoritmos estocásticos para problemas de otimização combinatória [2, 4, 3, 51]. De modo geral, esse tipo de gráfico exibe no eixo das ordenadas a probabilidade do algoritmo encontrar uma solução, no mínimo, igual a um valor *target*, em um dado tempo computacional exibido no eixo das abscissas.

O protocolo para gerar um *ttt-plot* é descrito a seguir. Deve-se escolher uma instância e um

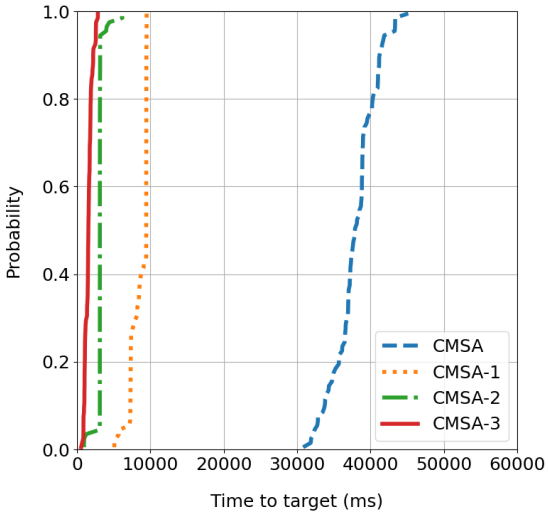
valor *target*, que pode ou não ser o valor ótimo. Em seguida, cada algoritmo a ser avaliado deve ser executado N vezes utilizando como critério de parada encontrar uma solução igual ou melhor que *target*. Ao final de cada uma, salva-se o tempo da execução. Para garantir que as execuções sejam independentes, os algoritmos deve ser inicializados com uma *seed* de números aleatórios diferente em cada uma. Ao finalizar as execuções, os tempos computacionais são ordenados em ordem crescente e, em seguida, são calculadas as probabilidades da seguinte forma: à i-ésima execução ordenada é atribuída a probabilidade $p_i = \frac{(1-\frac{1}{2})}{N}$. Por fim, são gerados e plotados pontos $y_i = (t_i, p_i), \forall i \in \{1, \ldots, N\}$.

A Figura 5.1 mostra os *ttt-plots* gerados a partir da execução do protocolo descrito. Cada gráfico exibe o *ttt-plot* relativo a uma instância, sendo 4 no total. As instâncias foram retiradas do dataset de Chimani et al. [11] do experimento anterior e o critério de seleção utilizado foram as instâncias mais difíceis para as quais a versão mais simples do algoritmo (`CMSA`) encontra um valor perto do ótimo.

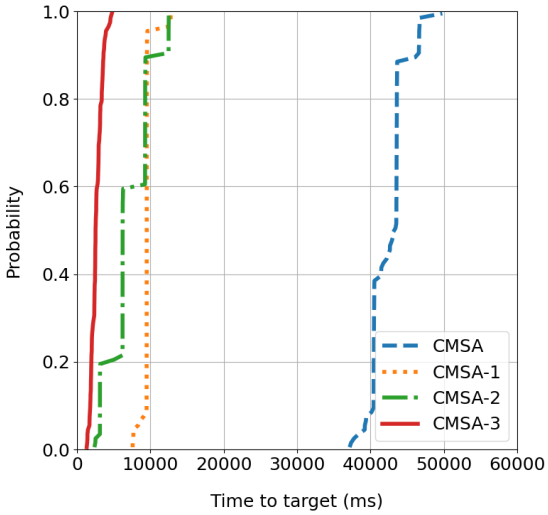
O gráficos 5.1 (a) e 5.1 (b) são resultantes da execução dos algoritmos nas instâncias binárias com $n \in \{40, 50\}$ e $m = 5000$. Os *targets* definidos para cada uma foram de 3337 e 3380, respectivamente ($\approx 0.08\%$ do ótimo para as duas instâncias). Já os gráficos 5.1 (c) e 5.1 (d) se referem às instâncias de proteínas com $n \in \{40, 50\}$ e $m = 5000$. Os *targets* definidos foram 4380 (0.2% do ótimo) e 4430 (0.33% do ótimo), respectivamente. Cada algoritmo foi executado 100 vezes em cada instância.

Analisando os 4 gráficos fica evidente que o `CMSA` apresenta o pior desempenho, seguido pelo `CMSA-1`, ambos sendo superados pelo `CMSA-2` e `CMSA-3`. Estes últimos apresentam desempenho bastante semelhante, com 100% de probabilidade de atingirem o *target* em menos de 15s nas 4 instâncias. Apesar dos resultados relativamente próximos, é notável a atuação do mecanismo de auto-adaptação em combinação com *callback* do CPLEX na convergência do `CMSA-3`, garantindo o alcance do target em, no máximo, 5 segundos.

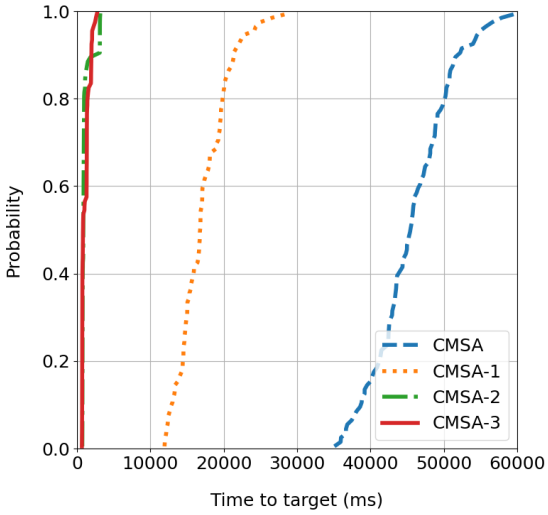
Os resultados das versões `CMSA-1` e `CMSA-2` também chamam bastante atenção. O desempenho do `CMSA-1` em comparação com o `CMSA` demonstra a eficácia da ação da estrutura tabu no CMSA puro. Nas 4 instâncias, o `CMSA` atinge o *target* em *no mínimo* 30 segundos, enquanto o `CMSA-1` o faz em *no máximo* 30 segundos. Vale notar que o `CMSA-2` aproxima-se bastante da origem do gráfico. Ainda, espeficiamente nos gráficos 5.1 (c) e 5.1 (d), o `CMSA-2` se distancia consideravelmente do `CMSA-1`, enquanto se aproxima dos resultados do `CMSA-3`. Tais observações indicam a importância da solução inicial para o encurtamento do tempo de execução do algoritmo.



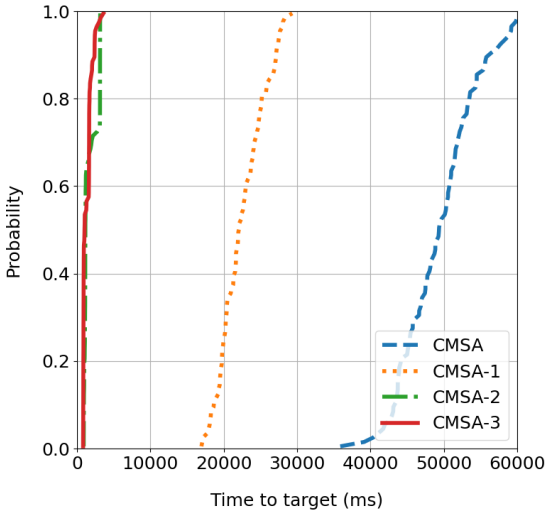
(a) 4-40-5000-1-0 (target 3337)



(b) 4-50-5000-1-0 (target 3380)



(c) 20-40-5000-1-0 (target 4380)



(d) 20-50-5000-1-0 (target 4430)

Figura 5.1: *ttt-plots* de 4 instâncias para 4 versões diferentes do CMSA, denominadas `CMSA`, `CMSA-1`, `CMSA-2` e `CMSA-3`. Fonte: Autor

6

Considerações Finais

Neste trabalho foi proposta uma *matheurística Adapt*-CMSA com Tabu para o Problema da Cadeia de Caracteres Mais Próxima (PCCP). Os experimentos realizados tiveram como objetivo comparar o desempenho do SA-CMSA TABU em instâncias de médio e grande porte com duas abordagens da literatura: *Simulated Annealing with Greedy Walk* (SAGW) e *Hybrid Discrete Particle Swarm Optimization* (HDPSO). Os resultados computacionais mostraram que o SA-CMSA TABU supera os outros dois algoritmos, encontrando o ótimo em cerca de 64% das instâncias de teste, enquanto o SAGW e HDPSO o encontram em aproximadamente 5% e 0,9%, respectivamente. Ainda, encontra soluções 1 unidade distante do valor encontrado pelo exato em 70% das que instâncias em que não encontra o ótimo. Além disso, o método proposto reduz significativamente a média de gap com relação ao valor encontrado pelo exato. Nas instâncias artificiais binárias, o SA-CMSA TABU tem média de gap aproximadamente 94% menor que a do SAGW e 99% menor que a do HDPSO. O mesmo padrão ocorre para as instâncias de DNA e proteínas, com média 91% e 95% menor que a do SAGW, respectivamente, e 99% menor que a do HDPSO em ambos os grupos.

Além disso, a contribuição dos diferentes componentes do algoritmo foi analisada através de gráficos de distribuição de probabilidade acumulada (*ttt-plots*). Foram selecionadas 4 instâncias de grande porte, as quais foram executadas em 4 versões distintas do algoritmo, cada uma representando uma evolução do algoritmo com relação à versão anterior. A plotagem dos gráficos mostram o impacto da lista tabu na convergência do algoritmo, o qual reduz o tempo para encontrar o *target* na metade em comparação com a aplicação pura do CMSA. Além disso, a adição da solução inicial, do mecanismo de auto-adaptação e do *callback* embutido no *solver* exato se mostram cruciais para uma melhora ainda mais significativa no desempenho do algoritmo.

Conclui-se, portanto, que a aplicação do *Adapt*-CMSA com Tabu ao PCCP foi extremamente satisfatória e possibilitou o avanço do estado da arte do problema, superando os algoritmos meta-heurísticos mais recentes encontrados na literatura. Como trabalhos futuros,

pretende-se testar algoritmos exatos diferentes como *solver* do CMSA. Ainda, uma etapa de busca local pode ser incorporada para tentar acelerar a convergência do algoritmo. Além disso, pretende-se realizar experimentos mais extensivos e direcionados à análise do desempenho do algoritmo.

Referências bibliográficas

[1] *Hybrid Metaheuristics*. Springer Berlin Heidelberg, 2013. ISBN 9783642306716. **DOI** 10.1007/978-3-642-30671-6. URL http://dx.doi.org/10.1007/978-3-642-30671-6.

[2] Renata M. Aiex, Mauricio G. C. Resende, Panos M. Pardalos, and Gerardo Toraldo. Grasp with path relinking for three-index assignment. *INFORMS Journal on Computing*, 17(2):224–247, 2005. **DOI** 10.1287/ijoc.1030.0059. URL https://doi.org/10.1287/ijoc.1030.0059.

[3] Renata M. Aiex, Mauricio G. C. Resende, and Celso C. Ribeiro. Ttt plots: a perl program to create time-to-target plots. *Optimization Letters*, 1(4):355–366, 2007. ISSN 1862-4480. **DOI** 10.1007/s11590-006-0031-4. URL https://doi.org/10.1007/s11590-006-0031-4.

[4] R.M. Aiex, S. Binato, and M.G.C. Resende. Parallel grasp with path-relinking for job shop scheduling. *Parallel Computing*, 29(4):393–430, 2003. ISSN 0167-8191. **DOI** https://doi.org/10.1016/S0167-8191(03)00014-0. URL https://www.sciencedirect.com/science/article/pii/S0167819103000140. Parallel computing in numerical optimization.

[5] Mehmet Anil Akbay, Albert López Serrano, and Christian Blum. A self-adaptive variant of cmsa: application to the minimum positive influence dominating set problem. *International Journal of Computational Intelligence Systems*, 15(1):44, 2022. **DOI** https://doi.org/10.1007/s44196-022-00098-1.

[6] Ambreen Ayub, Usman A Ashfaq, Sobia Idrees, and Asma Haque. Global consensus sequence development and analysis of dengue ns3 conserved domains. *BioResearch open access*, 2(5):392–396, Oct 2013. ISSN 2164-7844. **DOI** 10.1089/biores.2013.0022. URL https://doi.org/10.1089/biores.2013.0022.

[7] Maryam Babaie and Seyed Rasoul Mousavi. A memetic algorithm for closest string problem and farthest string problem. *2010 18th Iranian Conference on Electrical*

Engineering, pages 570–575, 2010. URL https://api.semanticscholar.org/CorpusID:13418173.

[8] Amir Ben-Dor, Giuseppe Lancia, R. Ravi, and Jennifer Perone. Banishing bias from consensus sequences. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching*, pages 247–261, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[9] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6): 4135–4151, 2011. ISSN 1568-4946. **DOI** https://doi.org/10.1016/j.asoc.2011.02.032. URL https://www.sciencedirect.com/science/article/pii/S1568494611000962.

[10] Christian Blum, Pedro Pinacho, Manuel López-Ibáñez, and José A. Lozano. Construct, merge, solve & adapt: A new general algorithm for combinatorial optimization. *Computers & Operations Research*, 68:75–88, 2016. ISSN 0305-0548. **DOI** https://doi.org/10.1016/j.cor.2015.10.014.

[11] Markus Chimani, Matthias Woste, and Sebastian Böcker. A closer look at the closest string and closest substring problem. In *2011 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 13–24, 01 2011. ISBN 978-1-61197-291-7. **DOI** 10.1137/1.9781611972917.2. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972917.2.

[12] Cheng-Chung Chou, Te-Tsui Lee, Chun-Houh Chen, Hsiang-Yun Hsiao, Yi-Ling Lin, Mei-Shang Ho, Pan-Chyr Yang, and Konan Peck. Design of microarray probes for virus identification and detection of emerging viruses at the genus level. *BMC Bioinformatics*, 7(1):232, 2006. ISSN 1471-2105. **DOI** 10.1186/1471-2105-7-232. URL https://doi.org/10.1186/1471-2105-7-232.

[13] Gerard Cohen, Mark Karpovsky, Mattson Jr, and James Schatz. Covering radius—survey and recent results. *Information Theory, IEEE Transactions on*, 31:328 – 343, 06 1985. **DOI** 10.1109/TIT.1985.1057043.

[14] Stephen A. Cook. The complexity of theorem-proving procedures. STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 9781450374644. **DOI** 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

[15] Lucas de O. Bastos, Luiz Satoru Ochi, Fábio Protti, Anand Subramanian, Ivan C. Martins, and Rian G. S. Pinheiro. Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization*, 31:347 – 371, 2014. **DOI** 10.1007/s10878-014-9756-7. URL https://doi.org/10.1007/s10878-014-9756-7.

[16] W. E. Duckworth, A. E. Gear, and A. G. Lockett. *What Operational Research Is and Does*, pages 1–15. Springer Netherlands, Dordrecht, 1977. ISBN 978-94-011-6910-3. **DOI** 10.1007/978-94-011-6910-3$_1$. URL https://doi.org/10.1007/978-94-011-6910-3_1.

[17] Simone Faro and Elisa Pappalardo. Ant-csp: An ant colony optimization algorithm for the closest string problem. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, pages 370–381, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11266-9.

[18] Paola Festa. On some optimization problems in molecular biology. *Mathematical biosciences*, 207:219–34, 07 2007. **DOI** 10.1016/j.mbs.2006.11.012.

[19] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997. **DOI** 10.1007/bf02679443.

[20] Robert S Garfinkel and George L Nemhauser. The set-partitioning problem: set covering with equality constraints. *Operations Research*, 17(5):848–856, 1969. **DOI** 10.1287/opre.17.5.848.

[21] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20:74–94, 08 1990. **DOI** 10.1287/inte.20.4.74.

[22] Fred Glover, Manuel Laguna, and Rafael Marti. *Tabu Search*, volume 16. 07 2008. **DOI** 10.1007/978-1-4615-6089-0.

[23] Fernando C Gomes, Cláudio N Meneses, Panos M Pardalos, and Gerardo Valdisio R Viana. A parallel multistart algorithm for the closest string problem. *Computers & Operations Research*, 35(11):3636–3643, 2008.

[24] Hermie J. M. Harmsen, Gerwin C. Raangs, Tao He, John E. Degener, and Gjalt W. Welling. Extensive set of 16s rrna-based probes for detection of bacteria in human feces. *Applied and Environmental Microbiology*, 68(6):2982–2990, 2002. **DOI** 10.1128/AEM.68.6.2982-2990.2002. URL https://journals.asm.org/doi/abs/10.1128/aem.68.6.2982-2990.2002.

[25] Bryant A. Julstrom. A data-based coding of candidate strings in the closest string problem. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, page 2053–2058, 2009. ISBN 9781605585055. **DOI** 10.1145/1570256.1570275. URL https://doi.org/10.1145/1570256.1570275.

[26] J. Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, 2003. ISSN 0890-5401. **DOI** https://doi.org/10.1016/S0890-5401(03)00057-9.

[27] Jehanara Korimbocus, Noël Scaramozzino, Bruno Lacroix, Jean Marc Crance, Daniel Garin, and Guy Vernet. Dna probe array for the simultaneous identification of herpesviruses, enteroviruses, and flaviviruses. *Journal of Clinical Microbiology*, 43(8): 3779–3787, 2005. **DOI** 10.1128/jcm.43.8.3779-3787.2005. URL https://journals.asm.org/doi/abs/10.1128/jcm.43.8.3779-3787.2005.

[28] Manuel Laguna. *Tabu Search*, pages 741–758. Springer International Publishing, Cham, 2018. ISBN 978-3-319-07124-4. **DOI** 10.1007/978-3-319-07124-4$_2$4. URL https://doi.org/10.1007/978-3-319-07124-4_24.

[29] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966. **DOI** 10.1287/opre.14.4.699. URL https://doi.org/10.1287/opre.14.4.699.

[30] Ming Li, Bin ma, and Lusheng Wang. On the closest string and substring problems. *Journal of the ACM*, 49, 03 2002. **DOI** 10.1145/506147.506150.

[31] Juntaek Lim, Seung Gu Shin, Seungyong Lee, and Seokhwan Hwang. Design and use of group-specific primers and probes for real-time quantitative pcr. *Frontiers of Environmental Science & Engineering in China*, 5(1):28–39, Mar 2011. ISSN 1673-7520. **DOI** 10.1007/s11783-011-0302-x. URL https://doi.org/10.1007/s11783-011-0302-x.

[32] Xiaolan Liu, Keqiang Fu, and Renxiang Shao. Largest distance decreasing algorithm for the closest string problem. *J. Inf. Comput. Sci.*, 1:287–292, 2004.

[33] Xiaolan Liu, med. Müller Holger, Zhifeng Hao, and Guan-Shiun Wu. A compounded genetic and simulated annealing algorithm for the closest string problem. *2008 2nd International Conference on Bioinformatics and Biomedical Engineering*, pages 702–705, 2008. URL https://api.semanticscholar.org/CorpusID:12406669.

[34] Xiaolan Liu, Shenghan Liu, Zhifeng Hao, and Holger Mauch. Exact algorithm and heuristic for the closest string problem. *Computers & Operations Research*, 38(11): 1513–1520, 2011. ISSN 0305-0548. **DOI** https://doi.org/10.1016/j.cor.2011.01.009. URL https://www.sciencedirect.com/science/article/pii/S0305054811000219.

[35] Xuan Liu, Hongmei He, and Ondrej Sykora. Parallel genetic algorithm and parallel simulated annealing algorithm for the closest string problem. 9 2006. URL

https://repository.lboro.ac.uk/articles/online_resource/Parallel_genetic_algorithm_and_parallel_simulated_annealing_algorithm_for_the_closest_string_problem/9403976.

[36] J J Lu, C L Perng, S Y Lee, and C C Wan. Use of pcr with universal primers and restriction endonuclease digestions for detection and identification of common bacterial pathogens in cerebrospinal fluid. *Journal of clinical microbiology*, 38(6):2076–2080, Jun 2000. ISSN 0095-1137. **DOI** 10.1128/JCM.38.6.2076-2080.2000. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC86732/.

[37] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. ISSN 2214-7160. **DOI** https://doi.org/10.1016/j.orp.2016.09.002. URL https://www.sciencedirect.com/science/article/pii/S2214716015300270.

[38] Vitorrio Maniezzo. *MATHEURISTICS: Algorithms and Implementations*. SPRINGER, 2021. **DOI** 10.1007/978-3-030-70277-9.

[39] Marcella Mcclure, T Vasi, and W Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Molecular biology and evolution*, 11:571–92, 08 1994. **DOI** 10.1093/oxfordjournals.molbev.a040162.

[40] Claudio Meneses, Zhaosong Lu, Carlos Oliveira, and Panos Pardalos. Optimal solutions for the closest string problem via integer programming. *Informs Journal on Computing - INFORMS*, 16:419–429, 11 2004. **DOI** 10.1287/ijoc.1040.0090.

[41] CN Meneses, PM Pardalos, MGC Resende, and A Vazacopoulos. Modeling and solving string selection problems. In *Second international symposium on mathematical and computational biology*, pages 54–64, 2005.

[42] Yassine Meraihi. *Qualité de service dans les réseaux sans fil maillés/vanet*. PhD thesis, 02 2017.

[43] Napoleao Nepomuceno, Plácido Pinheiro, and André Coelho. Tackling the container loading problem: A hybrid approach based on integer linear programming and genetic algorithms. pages 154–165, 04 2007. ISBN 978-3-540-71614-3. **DOI** 10.1007/978-3-540-71615-0$_1$4.

[44] Emily Brito de Oliveira, Mateus Silva Batista, and Rian Pinheiro. Uma abordagem híbrida cmsa para o problema da cadeia de caracteres mais próxima. In *Anais do Simpósio Brasileiro de Pesquisa Operacional*, São José dos Campos, 2023. Galoá.

[45] Alex Olvera, Marc Noguera-Julian, Athina Kilpelainen, Luis Romero-Martín, Julia G. Prado, and Christian Brander. Sars-cov-2 consensus-sequence and matching overlapping peptides design for covid19 immune studies and vaccine development. *Vaccines*, 8(3), 2020. ISSN 2076-393X. URL https://www.mdpi.com/2076-393X/8/3/444.

[46] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, volume 32. 01 1982. ISBN 0-13-152462-3. **DOI** 10.1109/TASSP.1984.1164450.

[47] Elisa Pappalardo, Domenico Cantone, and Panos M. Pardalos. A combined greedy-walk heuristic and simulated annealing approach for the closest string problem. *Optimization Methods and Software*, 29(4):673–702, 2014. **DOI** 10.1080/10556788.2013.833616.

[48] Adam M. Phillippy, Jacquline A. Mason, Kunmi Ayanbule, Daniel D. Sommer, Elisa Taviani, Anwar Huq, Rita R. Colwell, Ivor T. Knight, and Steven L. Salzberg. Comprehensive dna signature discovery and validation. *PLoS computational biology*, 3 (5):887–894, May 2007. ISSN 1553-734X. **DOI** 10.1371/journal.pcbi.0030098.

[49] Rian Gabriel S. Pinheiro, Ivan C. Martins, Fábio Protti, and Luiz Satoru Ochi. A matheuristic for the cell formation problem. *Optimization Letters*, 12(2):335–346, September 2017. ISSN 1862-4480. **DOI** 10.1007/s11590-017-1200-3. URL http://dx.doi.org/10.1007/s11590-017-1200-3.

[50] Jakob Puchinger and Günther R. Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In José Mira and José R. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, pages 41–53, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31673-2.

[51] Alberto Reyes and Celso C. Ribeiro. Extending time-to-target plots to multiple instances. *International Transactions in Operational Research*, 25(5):1515–1536, 2018. **DOI** https://doi.org/10.1111/itor.12507. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/itor.12507.

[52] Steven Roman. *Coding and Information Theory*. Springer-Verlag, Berlin, Heidelberg, 1992. ISBN 0387978127.

[53] Franz Rothlauf. *Optimization Methods*, pages 45–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. **DOI** 10.1007/978-3-540-72962-4$_3$. URL https://doi.org/10.1007/978-3-540-72962-4_3.

[54] Arun Sankaradoss, Suraj Jagtap, Junaid Nazir, Shefta E Moula, Ayan Modak, Joshuah Fialho, Meenakshi Iyer, Jayanthi S Shastri, Mary Dias, Ravisekhar Gadepalli, Alisha

Aggarwal, Manoj Vedpathak, Sachee Agrawal, Awadhesh Pandit, Amul Nisheetha, Anuj Kumar, Mahasweta Bordoloi, Mohamed Shafi, Bhagyashree Shelar, Swathi S Balachandra, Tina Damodar, Moses Muia Masika, Patrick Mwaura, Omu Anzala, Kar Muthumani, Ramanathan Sowdhamini, Guruprasad R Medigeshi, Rahul Roy, Chitra Pattabiraman, Sudhir Krishna, and Easwaran Sreekumar. Immune profile and responses of a novel dengue dna vaccine encoding an ediii-ns1 consensus design based on indo-african sequences. *Molecular therapy : the journal of the American Society of Gene Therapy*, 30(5):2058–2077, May 2022. ISSN 1525-0024. **DOI** 10.1016/j.ymthe.2022.01.013. URL https://doi.org/10.1016/j.ymthe.2022.01.013.

[55] A. F. M. Santos. Algoritmos heurísticos para a solução do problema da cadeia de caracteres mais próxima. Master's thesis, Centro Federal de Educação Tecnológica de Minas Gerais, 2018.

[56] Michael C Schatz. Biological data sciences in genome research. *Genome research*, 25 (10):1417–1422, 2015.

[57] Nikola Stojanovic, Piotr Berman, Deborah Gumucio, Ross Hardison, and Webb Miller. A linear-time algorithm for the 1-mismatch problem. In *Algorithms and Data Structures*, pages 126–135, 1997. ISBN 978-3-540-69422-9.

[58] M. Struik. *Covering codes*. Phd thesis 1 (research tu/e / graduation tu/e), Mathematics and Computer Science, 1994. URL https://doi.org/10.6100/IR425174.

[59] Anand Subramanian, Eduardo Uchoa, and Luiz Satoru Ochi. A hybrid algorithm for a class of vehicle routing problems. *Computers Operations Research*, 40(10):2519–2531, 2013. ISSN 0305-0548. **DOI** 10.1016/j.cor.2013.01.013. URL https://www.sciencedirect.com/science/article/pii/S030505481300021X.

[60] Andrew M. Sutton. Fixed-parameter tractability of crossover: Steady-state gas on the closest string problem. *Algorithmica*, 83(4):1138–1163, 2021. ISSN 1432-0541. **DOI** 10.1007/s00453-021-00809-8. URL https://doi.org/10.1007/s00453-021-00809-8.

[61] Stefan Voss, Thomas Stutzle, and Vittorio Maniezzo. *MATHEURISTICS: Hybridizing metaheuristics and mathematical programming*. Springer, 2009. **DOI** 10.1007/978-1-4419-1306-7.

[62] James D. Watson. The human genome project: Past, present, and future. *Science*, 248 (4951):44–49, 1990. **DOI** 10.1126/science.2181665. URL https://www.science.org/doi/abs/10.1126/science.2181665.

[63] Ya Jun Yu and Yong Lim. Design of linear phase fir filters in subexpression space using mixed integer linear programming. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54:2330 – 2338, 11 2007. **DOI** 10.1109/TCSI.2007.904599.