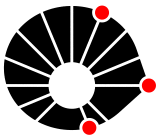
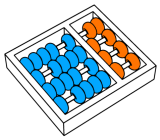




Heurísticas para Partição Comum Mínima de Strings

Leonardo de Sousa Rodrigues *Tiago de Paula Alves*
Gabriel Siqueira *Zanoni Dias*

Relatório Técnico - IC-PFG-23-61
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Heurísticas para Partição Comum Mínima de Strings

Leonardo de Sousa Rodrigues* Tiago de Paula Alves* Gabriel Siqueira*
Zanoni Dias*

Resumo

O problema de Partição Comum Mínima de Strings (MCSP) é utilizado na comparação de strings com aplicações em biologia computacional. Boas heurísticas têm grande valor para esse problema, já que ele foi provado ser NP-Difícil. Além de apresentar implementações para heurísticas conhecidas da literatura, desenvolvemos uma representação por grafo eficiente para instâncias do MCSP, reduzindo-o a um problema de permutação e permitindo a aplicação de algoritmos de otimização para buscar soluções. O *Particle Swarm Optimization* (PSO) foi adaptado para esta representação e foi capaz de não só melhorar significativamente o resultado das outras heurísticas utilizadas, mas também encontrar boas soluções de forma independente, mostrando-se uma meta-heurística promissora, principalmente para instâncias com poucas repetições de caracteres. Esse trabalho sugere a utilização da representação por grafo com outros métodos de otimização para o MCSP.

1 Introdução

Problemas de comparação de strings, que encontram o menor número de operações necessárias para formar uma string a partir de outra, têm diversas aplicações em biologia computacional, em processamento de texto e em compressão de arquivos [12]. O problema da partição comum mínima de strings (MCSP) se encaixa nessa classe de problemas, especificamente para o caso em que a única operação disponível é a reordenação de substrings.

Seja uma string A, com $|A|$ caracteres. Denotamos por A_i o i-ésimo caractere de A. O conjunto de caracteres distintos de A é chamado de *alfabeto* de A. Utilizamos o termo *rótulo* para se referir aos elementos do alfabeto e o termo caractere para se referir aos elementos da string [21, p. 17].

Definição 1 (*Ocorrência*)

A ocorrência de um rótulo α em uma string A é o número de cópias de α presentes em A.

Definição 2 (*Strings Balanceadas*)

Duas strings são ditas balanceadas se possuem o mesmo alfabeto e a ocorrência de todos os caracteres é igual nas duas strings.

*Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP.

Definição 3 (*Partição*)

Uma sequência de strings $\mathcal{P}$ é dita uma partição de uma string A se a concatenação dos elementos de $\mathcal{P}$ for igual a A. Chamamos as substrings de A em $\mathcal{P}$ de blocos. O tamanho de $\mathcal{P}$ é dado pelo seu número de blocos e denotado por $|\mathcal{P}|$.

Definição 4 (*Partição Comum*)

*Para uma partição $\mathcal{P}$ de A e outra partição $\mathcal{Q}$ de B, o par $(\mathcal{P},\mathcal{Q})$ é chamado de **partição comum** de A e B se $\mathcal{P}$ é uma permutação de $\mathcal{Q}$ [12]. O tamanho de uma partição comum $(\mathcal{P},\mathcal{Q})$ é dado por $|(\mathcal{P},\mathcal{Q})| = |\mathcal{P}| = |\mathcal{Q}|$.*

O **MCSP** (do inglês, *minimum common string partition*) consiste em encontrar uma partição comum de tamanho mínimo para duas strings balanceadas. Ao abordar o problema pela primeira vez, alguém poderia considerar uma solução direta: testar, para todas as possíveis configurações de separação em blocos de A, se é possível formar B a partir de uma permutação e, com isso, escolher as soluções de menor tamanho. É fácil notar, no entanto, que tal algoritmo não é polinomial, sendo inviável obter uma solução do problema dessa forma, mesmo com instâncias de menos de 50 caracteres. De fato, provou-se que o MCSP é um problema NP-Difícil, exceto para o caso em que cada caractere ocorre apenas uma vez em cada string [12]. Sendo assim, heurísticas são ótimos instrumentos para encontrar soluções boas para instâncias do problema.

Nas últimas décadas, diversos trabalhos fizeram progresso para encontrar melhores soluções e garantir aproximações para o MCSP. Em 2004, Chrobak, Kolman e Sgall [9] mostraram que a heurística gulosa tem aproximação em $\Omega(n^{0.43})$ e em $O(n^{0.69})$. Em 2005, Goldstein, Kolman e Zheng [12] provaram que o problema é NP-difícil, propondo uma 1.1037-aproximação para o 2-MCSP e uma 4-aproximação para o 3-MCSP. O k-MCSP é uma versão restrita do MCSP em que a ocorrência máxima da entrada é no máximo k. Ainda em 2005, Chen et al. [8] elaboraram o algoritmo SOAR com uma 1.5-aproximação para o 2-SMCSP (versão com sinais do problema). Em 2007, Cormode e Muthukrishnan [10] apresentaram uma aproximação em $O(\log n \log^* n)$ e Kolman e Waleń [18] mostraram uma $4k$-aproximação para o k-MCSP. Além disso, He [14] introduziu a ideia de explorar os rótulos de ocorrência 1 para melhorar a heurística gulosa e Kolman e Waleń [17] modificaram essa heurística para garantir aproximação em $O(k^2)$. Em 2010, Jiang et al. [15] propuseram um algoritmo parametrizado com tempo em $O^*((k!)^{2c})$, em que c é o tamanho da solução.

Em 2013, Bulteau e Komusiewicz [6] conseguiram um algoritmo parametrizado com tempo em $c^{21k^2}\text{poli}(n)$ e, no mesmo ano, Bulteau et al. [7] atingiram um algoritmo parametrizado com tempo em $O(k^{2c'} \cdot cn)$, em que c' é o número de blocos que contém pelo menos um caractere replicado. Em 2014, Goldstein e Lewenstein [13] sugeriram uma versão da heurística gulosa em tempo linear. No ano seguinte, Blum, Lozano e Davidson [2] formularam o problema utilizando programação linear inteira (PLI) e outra formulação com PLI foi feita por Blum e Raidl [3]. Em 2016, Blum et al. [4] construíram uma meta-heurística que utiliza soluções exatas de instâncias menores do problema. Esse algoritmo foi adaptado em 2018 para instâncias maiores [1].

Em 2016, Ferdous e Rahman [11] implementaram uma meta-heurística que se baseia no método da colônia de formigas para encontrar soluções para o MCSP, utilizando uma representação por grafo de substrings comuns das strings de entrada do problema. Recentemente, em 2021, Siqueira,

Alexandrino e Dias [22] apresentaram uma $2k$-aproximação para o problema.

Nas seções desse relatório, primeiramente mostraremos e explicaremos heurísticas diretas e determinísticas para o MCSP: a de combinação e a gulosa. Depois, discutiremos uma representação por grafo para as entradas do problema, que permite a aplicação de algoritmos de otimização. Em seguida, mostraremos detalhadamente uma implementação do *Particle Swarm Optimization* utilizado em conjunto com a representação desenvolvida. Por fim, discutiremos os resultados obtidos e o tempo de execução das heurísticas.

2 Heurísticas Iniciais

Como uma primeira abordagem para a resolução do MCSP, apresentaremos duas heurísticas diretas e determinísticas, que utilizam alguma característica do problema para tentar encontrar soluções com boa qualidade: a heurística de combinação e a heurística gulosa.

2.1 Heurística de Combinação

A heurística de combinação consiste em um algoritmo que inicia com uma solução trivial do problema (blocos de um caractere cada) e tenta agrupar blocos mantendo uma solução válida. A implementação escolhida consiste em analisar os blocos das strings da esquerda para a direita e agrupá-los sempre que possível, como mostra o Algoritmo 1. A escolha arbitrária de quais blocos serão combinados, no entanto, compromete a qualidade da solução gerada.

Algoritmo 1: Heurística de combinação.

COMBINAÇÃO(A, B)
1 $B_A \leftarrow$ blocos unitários de A
2 $B_B \leftarrow$ blocos unitários de B
3 **para cada** par de blocos (b_1, b_2) consecutivos em B_A e em B_B **faça**
4 $\quad B_A \leftarrow B_A$ com b_1 e b_2 combinados
5 $\quad B_B \leftarrow B_B$ com b_1 e b_2 combinados
6 **devolva** (B_A, B_B)

2.1.1 Análise de Singletons

Definição 5 (*Singleton*)

Um caractere de uma string é dito singleton se a ocorrência de seu rótulo na string é 1.

Caracteres singletons têm valor especial na resolução do MCSP: uma substring que contém tal caractere em A não pode ter mais de uma substring equivalente em B. Com isso, é possível priorizar a combinação de blocos que contêm singletons e, com isso, melhorar a tomada de decisão do algoritmo

e seus resultados. O Algoritmo 2 mostra como adaptar o procedimento de combinação de blocos. Chamaremos essa modificação de combinação-S.

Algoritmo 2: Heurística de combinação com análise de singletons.

COMBINAÇÃO-S(A,B)

1 $B_A \leftarrow$ blocos unitários de A
2 $B_B \leftarrow$ blocos unitários de B
3 **para cada** par de blocos (b_1,b_2) consecutivos em B_A e em B_B tal que b_1 e b_2 possuam singletons **faça**
4 $\quad B_A \leftarrow B_A$ com b_1 e b_2 combinados
5 $\quad B_B \leftarrow B_B$ com b_1 e b_2 combinados
6 **para cada** par de blocos (b_1,b_2) consecutivos em B_A e em B_B tal que b_1 ou b_2 possuam singletons **faça**
7 $\quad B_A \leftarrow B_A$ com b_1 e b_2 combinados
8 $\quad B_B \leftarrow B_B$ com b_1 e b_2 combinados
9 **para cada** par de blocos (b_1,b_2) consecutivos em B_A e em B_B **faça**
10 $\quad B_A \leftarrow B_A$ com b_1 e b_2 combinados
11 $\quad B_B \leftarrow B_B$ com b_1 e b_2 combinados
12 **devolva** (B_A,B_B)

2.2 Heurística Gulosa

Uma outra estratégia para encontrar soluções do MCSP é a gulosa. Tal algoritmo consiste em iterativamente escolher blocos grandes que coincidam entre as duas strings e marcá-los. Em cada iteração, escolhe-se a maior substring comum entre S e P que não coincida com blocos já marcados em nenhuma das strings. Os dois novos blocos correspondentes são então marcados. O algoritmo progride dessa forma até que todos os caracteres das strings pertençam a blocos marcados, formando então uma partição comum.

2.2.1 Maior Substring Comum

O problema de encontrar a maior substring comum entre duas strings pode ser resolvido com a ajuda de uma árvore de sufixos. Tal árvore é uma versão especializada de uma *Radix Tree* contendo todos os sufixos de cada string e algum marcador referente a string original.

Com essa representação, o problema da maior substring comum se transforma em um problema de encontrar o ancestral comum mais recente entre duas folhas de origens distintas na árvore. Esse ancestral comum representa o maior prefixo comum entre todos os sufixos das duas strings, sendo portanto a maior substring comum entre elas.

A árvore de sufixos pode ser adaptada também para manter substrings de múltiplas string distintas, permitindo a implementação eficiente da maior substring comum em uma coleção de strings. Esse adaptação não foi implementada nesse trabalho.

Algoritmo 3: Heurística gulosa.

GULOSA(A,B)

```
1  B_A ← sequência com A como único elemento
2  B_B ← sequência com B como único elemento
3  enquanto existem caracteres pertencentes a blocos não marcados faça
4      x ← maior substring comum a B_A e B_B nos blocos não marcados
5      quebre um bloco de B_A que contém x criando um novo bloco marcado com apenas x
6      quebre um bloco de B_B que contém x criando um novo bloco marcado com apenas x
7  devolva (B_A, B_B)
```

3 Representação por Grafo

Outra forma de construir soluções boas para o MCSP é utilizando meta-heurísticas, i.e., algoritmos de otimização que utilizam aleatorização em conjunto com busca local [23, p. 4]. Antes de aplicar algum dos vários algoritmos conhecidos, no entanto, é necessário representar as instâncias do problema utilizando uma estrutura de dados eficiente e adequada para tais procedimentos. Encontrar uma boa representação para problemas complexos, como esse, é uma etapa desafiadora, mas que pode simplificar muito a resolução.

Desenvolvemos uma representação baseada em um grafo de substrings comuns [11], adaptando a representação para arestas múltiplas e ignorando as substrings de tamanho unitário.

Definição 6 (*Grafo de Substrings Comuns*)

Dado um par de strings balanceadas A e B de tamanho $n = |A| = |B|$, temos o grafo de substrings comuns $G_{A,B}(V,E,\varphi)$ de A em B onde:

a) os vértices são dados pelos índices da string A:

$$V = \{1,\ldots,n\} = I_n$$

b) as arestas representam blocos não-unitários (mais de um caractere) e suas posições como substrings de A e B:

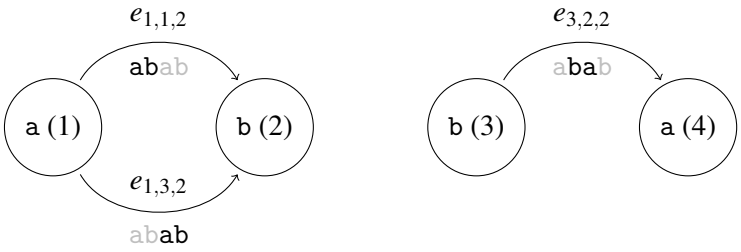
$$E = \left\{ e_{p,q,k} \;\middle|\; A_p \ldots A_{p+k-1} = B_q \ldots B_{q+k-1} \;\text{ e }\; k \geq 2 \right\}$$

c) as arestas conectam os caracteres inicial e final de A no bloco:

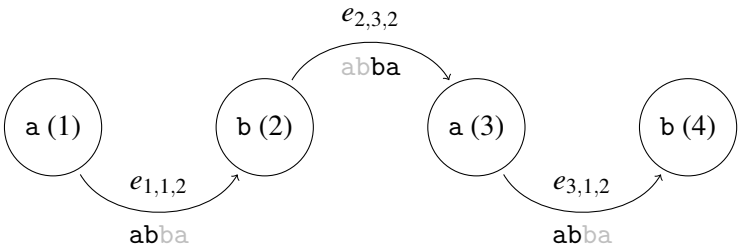
$$\varphi\left(e_{p,q,k}\right) = \{p, p+k-1\}$$

Note que cada par de bloco representado por uma aresta pode ser utilizado na construção de uma partição comum entre as strings. Arestas paralelas indicam que existe mais de uma substring em B igual à substring de A formada pelos caracteres entre suas extremidades. Um exemplo da representação pode ser visto na Figura 1.

É importante ressaltar que blocos de apenas 1 caractere não são considerados. Como as strings de entrada são balanceadas e os blocos das arestas são idênticos, partindo de uma partição comum



(a) Grafo representando a instância (abba, abab).



(b) Grafo representando a instância (abab, abba).

Figura 1: Grafo representando duas instâncias do MCSP, sendo ambas com as mesmas strings mas em ordens diferentes. Os vértices do grafo são índices da primeira string da instância e, em cada aresta do grafo, está evidenciada a substring equivalente que pertence à segunda string.

incompleta, podemos completá-la com todos os caracteres não contidos na partição comum como blocos unitários. Não considerar tais blocos reduz consideravelmente o número de arestas da representação, e consequentemente possibilita algoritmos mais eficientes.

3.1 Construindo Partições

A partir de uma sequência ordenada das arestas da representação de uma instância do MCSP, podemos construir partições da seguinte forma: para cada aresta, na ordem dada, incluímos os dois blocos à partição comum se nenhum deles coincide com caracteres contidos em blocos que já foram incluídos. Ao fim do processo, cria-se blocos unitários para todos os caracteres não contidos em um bloco. Dessa forma, uma permutação do conjunto de arestas do grafo representa uma única partição comum entre as strings de entrada do problema.

É válido ressaltar duas propriedades dessa representação. Primeiramente, nem todas as possíveis partições possuem uma permutação correspondente, já que os blocos unitários são omitidos. Tais partições, no entanto, não são as mais interessantes para o MCSP, já que buscamos partições menores.

Em segundo lugar, note que a partição comum formada por uma permutação das arestas não é única: podemos trocar de posição arestas vizinhas que não são utilizadas, de forma que o mesmo resultado é obtido.

3.2 Aplicação da Representação

Com a construção dessa representação, agora o MCSP se reduz a encontrar uma maneira de ordenar as arestas de forma que a partição comum resultante seja a menor possível. Isso significa que o reduzimos a um problema de permutação. Algoritmos de otimização que atuam em um espaço de busca contínuo podem ser utilizados para esse tipo de problema aplicando um sistema de pesos que assumem valores reais [19, p. 661]. Relacionamos o vetor de arestas com um vetor de pesos de mesmo tamanho, de forma que a ordenação das arestas pelos valores dos pesos correspondentes forma a permutação correspondente àqueles valores. Como cada peso é uma componente no espaço de busca, podemos nos referir ao vetor de pesos também como posição (e à sua variação, como velocidade).

4 Particle Swarm Optimization

Utilizando a representação por grafo desenvolvida, optamos por implementar o *Particle Swarm Optimization* (PSO). Esse algoritmo foi descoberto durante simulações de modelos sociais de animais [16] e baseia-se no comportamento de enxames e bandos de aves – mas também de cardumes ou até grupos humanos [23, p. 7]. Cada agente busca localmente em seus arredores por posições de qualidade, participando também da comunicação do grupo para que todos saibam qual é a melhor encontrada até o momento. O PSO ganhou popularidade nas últimas duas décadas devido ao balanço que proporciona entre a eficiência da busca por soluções e a facilidade de implementação e de adaptação ao problema em que é aplicado [19, p. 640].

De forma mais prática, o algoritmo consiste em um conjunto de partículas em que cada uma possui uma posição atual e mantém gravada a melhor posição que encontrou até o momento. O enxame, que contém as partículas, também mantém a melhor posição global encontrada. A cada iteração, a posição de uma partícula é atualizada de acordo com as componentes escolhidas, como explicado na Seção 4.2.

Para aplicar ao contexto do MCSP, iniciamos o algoritmo construindo a representação por grafo da instância e obtendo o vetor de arestas. Posições iniciais para a quantidade de partículas desejadas são geradas, conforme explicado na Seção 4.1. Para medir a qualidade da solução referente a uma partícula, utilizamos a posição como vetor de pesos para obter uma ordenação das arestas do grafo. Essa ordenação é analisada para avaliar quantos blocos existe naquela solução, sem necessariamente gerar as partições correspondentes, já que isso apenas aumentaria o tempo de execução. Finalmente calculamos a função objetivo, a ser maximizada, dada pela diferença entre o número de caracteres da string e o número de blocos da partição. Ao fim do número de iterações desejado, utilizamos a melhor posição global para construir a solução do problema.

4.1 Posições Iniciais

Foram implementadas diferentes formas de gerar posições iniciais para as partículas visando diversificar o enxame, cobrindo ao máximo o espaço de busca, mas também incluindo soluções boas como ponto de partida para algumas partículas. As componentes das posições, apesar de não serem limitadas durante a execução, são inicializadas no intervalo $[-1, 1]$.

Uma das formas de incluir boas soluções é utilizar uma partição comum resultante da aplicação de outra heurística para gerar um vetor de pesos correspondente. Isso é feito verificando quais arestas do grafo são compatíveis com a partição comum dada, i.e., quais arestas têm seu bloco naquela partição. Feito isso, geramos um peso para cada aresta compatível no intervalo $[-1,0)$ e para cada não compatível no intervalo $[0,1]$, de forma que, ao serem ordenadas por esses pesos, a partição construída terá todos os blocos não-unitários da partição inicial, já que as arestas compatíveis serão consideradas antes das outras. Os pesos são gerados aleatoriamente no intervalo dado, permitindo a geração de diversas posições iniciais a partir da mesma partição comum.

Outra forma de propiciar posições iniciais de qualidade é incluindo ordenações do vetor de arestas por tamanho do bloco, obtendo partições comuns similares à heurística gulosa. Para isso, são gerados pesos aleatórios no intervalo $[-1,1]$ e arranjados de forma a priorizar as arestas com maiores blocos.

Por fim, a geração totalmente aleatória dos pesos uniformemente no intervalo $[-1,1]$, apesar de ingênua, gerou bons resultados por si só e também mostrou-se útil quando combinada às outras formas de construção de posições iniciais. De forma geral, consolidamos a criação das posições iniciais do enxame utilizando os procedimentos descritos:

a) A partir do resultado da heurística gulosa
b) A partir do resultado da heurística de combinação-S
c) A partir da ordenação das arestas por tamanho do bloco
d) Geração totalmente aleatória

Para cada partícula, uma das formas é escolhida com probabilidade proporcional a um peso designado a ela. Utilizaremos a ordem das formas de geração para simplificar a notação que representa a distribuição das partículas. Notaremos $D = (1,2,0,2)$, por exemplo, para indicar que uma distribuição de probabilidades foi usada com 20% de chance do resultado da heurística gulosa ser utilizado, 40% de chance do resultado da heurística combinação-S ser utilizado, 0% de chance da ordenação por tamanho do bloco ser utilizada e 40% da geração totalmente aleatória.

4.2 Movimento das Partículas

No início de cada iteração do algoritmo, as partículas devem se movimentar pelo espaço de busca para buscar melhores soluções. Esse movimento pode ser composto por diferentes componentes. Nas origens do PSO, utilizava-se duas componentes: uma em direção à melhor solução encontrada pela própria partícula (melhor individual) e outra em direção à melhor solução encontrada pela vizinhança da partícula. A vizinhança pode ser definida de diversas maneiras, como por distância ou como apenas uma vizinhança global [5].

Experimentos posteriores com o PSO consideraram diferentes componentes para atualização das posições, como a implementação de um sistema de inércia e um limite máximo para a velocidade da partícula [20].

Nossa implementação do algoritmo aqui apresentada para o MCSP utiliza uma combinação das componentes originais, com uma vizinhança global, e um comportamento estocástico [19, p. 642]. Seja $\vec{p}_j$ a posição de uma partícula na iteração j, $\vec{p}_L$ a sua melhor posição até agora e $\vec{p}_G$ a melhor posição encontrada pelo enxame. Define-se estaticamente k_L, k_G e k_E, como parâmetros do algoritmo. Se $R_L \sim \text{Uniforme}(0,k_L)$, $R_G \sim \text{Uniforme}(0,k_G)$ e se $\vec{R}_E$ representa um vetor de

componentes geradas uniformemente de forma aleatória no intervalo $[-k_E, k_E]$, então

$$\vec{p}_{j+1} = \vec{p}_j + (\vec{p}_L - \vec{p}_j) \cdot R_L + (\vec{p}_G - \vec{p}_j) \cdot R_G + \vec{R}_E$$

4.3 Exemplo de Execução

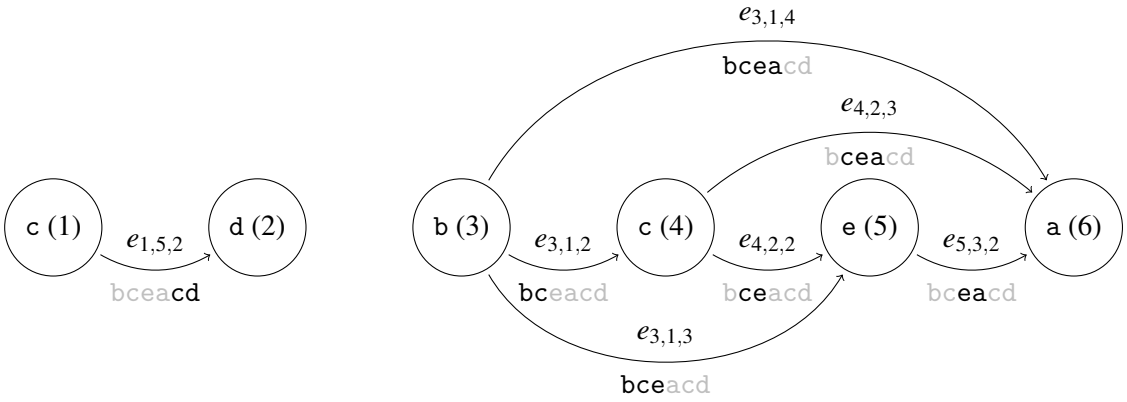


Figura 2: Grafo representando a instância (`cdbcea`, `bceacd`).

Para demonstrar o funcionamento do PSO, vejamos como uma entrada pequena do problema é abordada. Utilizamos para isso a instância (`cdbcea`, `bceacd`). O primeiro passo é criar a representação por grafo, que pode ser vista na Figura 2. A partir dela, podemos extrair o vetor de arestas, de tamanho 7.

Neste pequeno exemplo, utilizamos uma única partícula com 5 iterações. A posição inicial foi gerada de forma totalmente aleatória e escolhemos $k_L = 0.1$, $k_G = 0.1$ e $k_E = 2$. A Tabela 1 mostra o vetor de arestas em sua ordenação original e, para as 5 iterações do algoritmo, as componentes da posição da partícula e a partição correspondente ao vetor de arestas ordenado utilizando as componentes como pesos.

Componentes da posição (pesos)							Partição comum resultante
$e_{1,5,2}$	$e_{3,1,2}$	$e_{3,1,3}$	$e_{3,1,4}$	$e_{4,2,2}$	$e_{4,2,3}$	$e_{5,3,2}$	
-0.043	-0.069	0.186	-0.901	-0.957	-0.409	-0.478	`[cd,b,ce,a],[b,ce,a,cd]`
-0.840	0.985	-0.463	-0.416	-0.969	0.993	0.190	`[cd,b,ce,a],[b,ce,a,cd]`
0.763	-0.192	0.966	0.088	-1.483	-0.062	0.626	`[cd,b,ce,a],[b,ce,a,cd]`
-0.452	0.195	0.252	-1.924	0.349	-0.754	0.211	`[cd,bcea],[bcea,cd]`
-0.443	1.671	-0.979	-1.739	-0.514	-2.181	-0.867	`[cd,bce,a],[bce,a,cd]`

Tabela 1: Posições de uma partícula e respectiva partição comum durante 5 iterações da execução do PSO na instância (`cdbcea`, `bceacd`). Cada componente da posição é utilizada como peso para ordenar as arestas do grafo e encontrar diferentes partições comuns.

Note que as primeiras arestas a serem consideradas são as relacionadas aos menores pesos, já que a ordenação é feita por ordem crescente deles. Ao observar o grafo da Figura 2, podemos esperar que uma posição que resulta em uma partição comum de tamanho pequeno (`[cd,bcea]`, `[bcea,cd]`) deve ter a componente relacionada à aresta $e_{3,1,4}$ com valor menor às relacionadas a outras arestas coincidentes. De fato, a quarta iteração da Tabela 1, que contém essa exata partição comum resultante, tem como menor componente aquela relacionada à aresta $e_{3,1,4}$.

Na Figura 3 podemos ver também, para outra execução, o comportamento das partículas ao longo de algumas iterações na busca por um solução ótima. Mesmo iniciando em uma região com soluções piores, as partículas conseguem chegar em regiões com resultados melhores com o avanço das iterações.

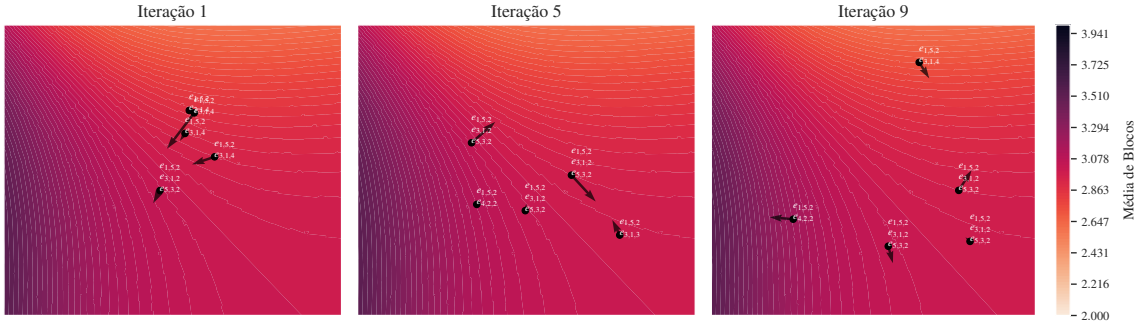


Figura 3: Representação gráfica de algumas iterações do PSO na instância (`cdbcea`, `bceacd`) com 5 partículas. O espaço de busca original é composto de sete dimensões, uma para cada aresta, mas foi reduzido a duas dimensões para ter uma visualização mais clara. As cores indicam as média de blocos resultantes nas soluções daquela região antes da redução de dimensionalidade. As arestas selecionadas do grafo (Figura 2) em cada partícula são apresentadas ao lado dela.

5 Resultados e Discussão

A fim de avaliar os resultados e a eficiência das heurísticas implementadas, criamos um processo para geração de instâncias do problema. Para criar um par de string balanceadas, escolhe-se: o número de rótulos replicados r (ocorrência maior que 1), o número de singletons s e o tamanho das strings n, sendo que $n \geq 2 \cdot r + s$. Uma string de inteiros é construída concatenando: duas cópias de cada inteiro no intervalo $[1, r]$, uma cópia de cada inteiro no intervalo $[r+1, r+s]$ e $n-2 \cdot r+s$ inteiros escolhidos aleatoriamente no intervalo $[1, r]$. Para obter o par de strings de entrada para o MCSP, duas permutações totalmente aleatórias da string formada são utilizadas.

Analisaremos na Seção 5.1 o tempo de execução dos algoritmos nos diferentes cenários, visando demonstrar as situações que favorecem algumas das heurísticas. Em seguida, na Seção 5.2, discutiremos o tamanho das partições resultantes para avaliar de fato cada um dos algoritmos. Para todos os testes realizados, exceto quando explicitado, fora utilizadas pelo menos 3000 diferentes instâncias do problema.

5.1 Tempo de Execução

O tempo de execução de cada um dos algoritmos é bem diferente, como pode-se observar na Tabela 2. Enquanto o tempo para aplicar as heurísticas de combinação é da ordem de microssegundos e para a gulosa da ordem de milissegundos, o PSO pode demorar vários segundos para ter sua execução finalizada, dependendo de seus parâmetros e das instâncias.

É válido ressaltar também a diferença entre a variação do tempo de execução das heurísticas com os parâmetros de geração das strings. Enquanto o fator principal pelo aumento do tempo para os outros algoritmos é o tamanho das strings de entrada, o PSO é substancialmente mais rápido para instâncias com maior número de rótulos e, consequentemente, menor ocorrência média de seus rótulos. Essa dependência será mais explorada na Seção 5.1.1.

n	r	s	Combinação [µs]	Combinação-S [µs]	Gulosa [ms]	PSO [s]
60	5	5	165	594	4.75	17.5
60	15	10	211	563	8.91	1.21
80	10	15	331	1140	17.4	5.79
80	20	30	401	1200	23.9	0.55

Tabela 2: Comparação de tempo de execução das diferentes heurísticas, relacionando cada conjunto de parâmetros de geração de strings (n, r e s) ao tempo gasto por cada um dos algoritmos. Os valores de tempo foram obtidos através da média de execuções em diferentes instâncias, com a análise estatística realizada através da biblioteca *Criterion*[1]. O PSO foi executado com $k_E = 0.5$, $k_L = 0.1$, $k_G = 0.1$, $D = (1,3,3,3)$, 200 partículas e 100 iterações.

5.1.1 Particle Swarm Optimization

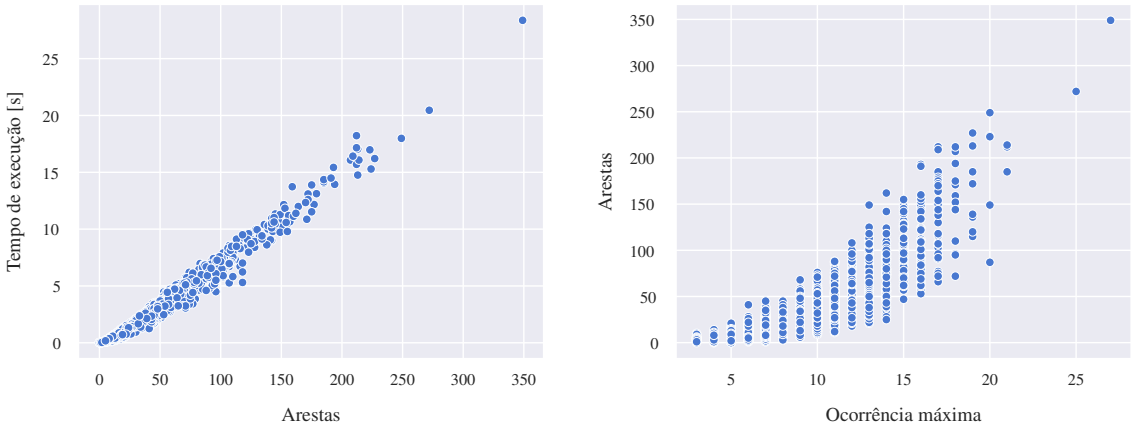
Analisando de forma mais detalhada o tempo de execução do PSO, espera-se que este dependa linearmente do número de arestas da representação por grafo da entrada do problema, já que praticamente todas as operações são feitas sobre vetores desse tamanho. De fato, a Figura 4a deixa clara essa dependência.

O número de arestas do grafo, por sua vez, claramente tende a crescer com o tamanho das strings, mas ele está vinculado ainda mais à ocorrência dos caracteres: quanto maior a ocorrência dos caracteres de um bloco, maior tende a ser o número de blocos iguais a ele na outra string de entrada, formando assim mais arestas. A Figura 4b mostra essa relação utilizando a ocorrência máxima das strings. Isso indica que o PSO é eficiente para casos em que os caracteres são replicados um número pequeno de vezes.

5.2 Tamanho das Partições

Para avaliar o resultado da aplicação das heurísticas de forma independente do tamanho da string, definimos uma forma de pontuação, de forma que uma partição comum com apenas blocos unitários

[1] *Criterion: Robust, reliable performance measurement and analysis*. Versão 1.6.3.0. URL: https://hackage.haskell.org/package/criterion



(a) Tempo de execução do PSO em função do número de arestas do grafo.

(b) Número de arestas do grafo em função da ocorrência máxima das strings.

Figura 4: Análise do tempo de execução do PSO. Note que o tempo depende linearmente do número de arestas da representação por grafo da instância de entrada, como esperado. Além disso, a grandeza que melhor permite prever o tempo de execução do algoritmo é a ocorrência máxima dos caracteres da string. As strings foram geradas usando $5 \leq r \leq 10$, $5 \leq s \leq 30$, $n \leq 80$. O PSO foi executado utilizando $k_E = 2$, $k_L = 0.02$, $k_G = 0.02$, $D = (3,1,1,1)$, 10 partículas e 1000 iterações.

tem pontuação 0 e uma partição comum com apenas um bloco recebe pontuação 1. Note que raramente a solução ótima do MCSP recebe pontuação 1: isso ocorre apenas se as strings forem iguais.

Definição 7 (*Pontuação*)

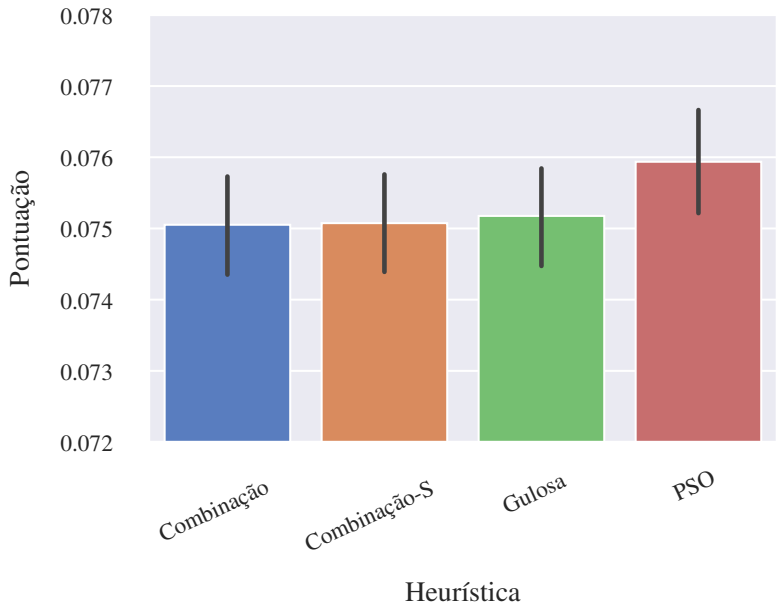
A pontuação de uma partição comum $(\mathcal{P}, \mathcal{Q})$ *de duas strings* A *e* B *é um valor no intervalo* $[0,1]$ *que indica quão pequena* $(\mathcal{P}, \mathcal{Q})$ *é em relação ao tamanho de* A *e* B*. O valor da pontuação é dado por*

$$S(\mathcal{P}, \mathcal{Q}) = \frac{|A| - |(\mathcal{P}, \mathcal{Q})|}{|A| - 1}$$

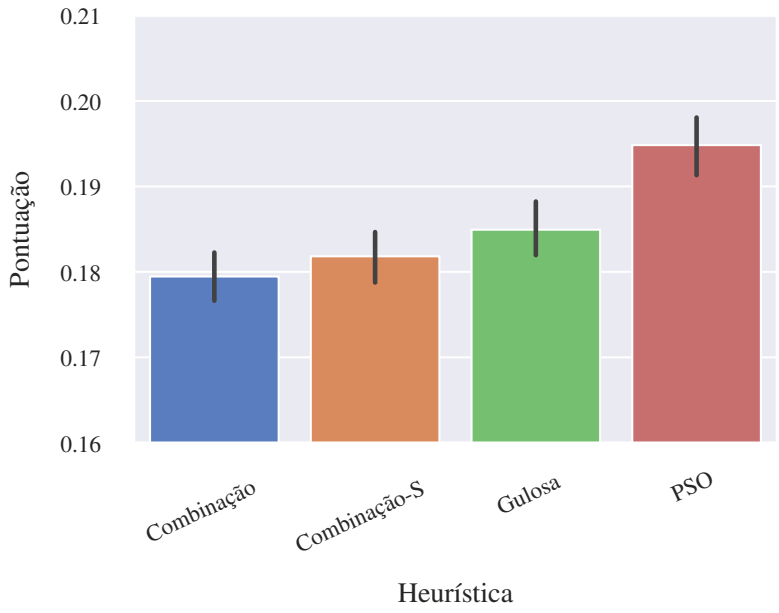
A Figura 5 mostra a pontuação média das heurísticas para diferentes parâmetros de geração de strings. Em ambos os casos, o PSO conseguiu melhorar ligeiramente o resultado obtido pelos outros algoritmos (já que utiliza os outros resultados como posições iniciais). Uma análise mais específica da diferença obtida entre o PSO e as outras heurísticas pode ser vista na Figura 6.

5.2.1 PSO com Posições Iniciais Totalmente Aleatórias

Até aqui analisamos o desempenho do PSO utilizando algumas das posições iniciais criadas a partir de resultados de outras heurísticas, de forma que seu resultado foi sempre no mínimo tão bom quanto ao da melhor heurística entre a gulosa e a combinação-S. A Figura 7 mostra uma comparação que inclui também o PSO com todas as posições iniciais sendo geradas de forma totalmente aleatória.



(a) Strings geradas com $r = 50$, $5 \leq s \leq 10$ e $n \leq 220$.



(b) Strings geradas com $5 \leq r \leq 10$, $5 \leq s \leq 30$ e $n \leq 80$.

Figura 5: Comparação da pontuação média das heurísticas com diversos conjuntos de parâmetros de geração de strings. Em ambos os casos, o PSO foi utilizado com $k_E = 0.5$, $k_L = 0.1$, $k_G = 0.1$, $D = (1,3,3,3)$, 200 partículas e 100 iterações.

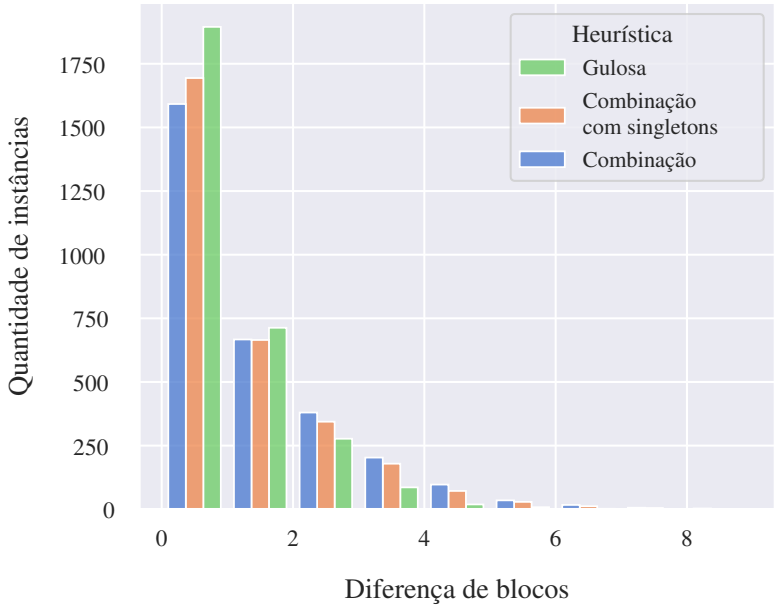


Figura 6: Comparação entre o número de blocos resultantes da aplicação do PSO e das outras heurísticas. O gráfico mostra a distribuição da diferença entre o número de blocos obtidos pelo PSO e uma dada heurística na mesma instância do MCSP. Note que o resultado mais comum é o empate. Strings geradas com $5 \leq r \leq 10$, $5 \leq s \leq 30$ e $n \leq 80$. PSO executado com $k_E = 0.5$, $k_L = 0.1$, $k_G = 0.1$, $D = (1,3,3,3)$, 200 partículas e 100 iterações.

Apesar de obter uma pontuação média ligeiramente menor que o algoritmo que inclui as outras formas de gerar posições iniciais, essa variação mais simples também foi capaz de obter melhores pontuações que as heurísticas gulosa e combinação-S, o que demonstra a eficácia do algoritmo, mesmo quando utilizado sem o auxílio das outras heurísticas.

5.2.2 PSO com Combinação

O PSO não garante que a partição comum resultante é maximal em relação à combinação de blocos adjacentes. Dessa forma, pode-se aplicar a heurística de combinação ao resultado obtido numa tentativa de melhorá-lo. Após alguns testes, concluímos que na prática são raríssimos os casos em que há alteração do tamanho da partição comum, mas como o tempo de execução da heurística de combinação é desprezível quando comparado ao do PSO, essa seria uma boa prática para garantir um resultado maximal.

Outra possibilidade seria aplicar a heurística de combinação às partições comuns formadas por cada partícula em cada iteração antes de medir o tamanho, adicionando uma etapa à função objetivo da otimização. Testes com essa alteração mostraram que, para algumas instâncias, resultados ligeiramente melhores são obtidos, mas para algumas outras o tamanho da partição comum resultante é maior. Isso acontece porque, ao alterar a forma com que as partículas são avaliadas, as melhores posições individual e global são outras e, consequentemente, as partículas seguem por caminhos

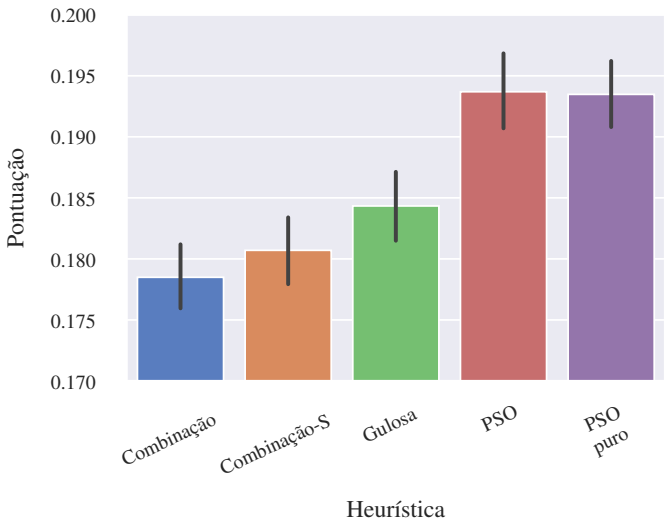


Figura 7: Comparação entre a pontuação média das heurísticas, incluindo o PSO com posições inicias totalmente aleatórias. Apesar de haver uma ligeira diferença da pontuação média do PSO utilizando posições iniciais das outras heurísticas, o PSO com apenas posições iniciais aleatórias apresentou pontuação maior que as outras heurísticas. Strings geradas com $5 \leq r \leq 10$, $5 \leq s \leq 30$ e $n \leq 80$. PSO executado com $k_E = 2$, $k_L = 0.1$, $k_G = 0.1$, 200 partículas, 100 iterações e, para as execuções incluindo posições iniciais obtidas de outras formas, $D = (1,3,3,3)$.

diferentes. Além disso, nesse caso, o tempo de execução é impactado significativamente, já que a heurística de combinação é executada para cada partícula em cada iteração.

5.2.3 Outros Algoritmos da Literatura

Além de comparar as heurísticas implementadas entre si, executamos também outros algoritmos desenvolvidos para o MCSP. Em um primeiro momento, utilizamos outras implementações das heurísticas gulosa e combinação-S disponíveis [22]. Ao aplicar as nossas e as outras implementações às mesmas instâncias do problema, obtivemos resultados idênticos.

Comparamos também, para um conjunto restrito de instâncias pequenas, os resultados obtidos por um algoritmo de parâmetro fixo (FPT), que é exato para o MCSP, i.e., sempre retorna uma solução ótima. A Figura 8 mostra que nenhuma das heurísticas acha de forma consistente a solução ótima para o problema, mas têm pontuações próximas a isso. A precisão desse conjunto de resultados é reduzida devido às restrições de quantidade e tamanho das instâncias, causadas pelo longo tempo necessário para executar o FPT. Ainda assim, é possível observar a tendência para cada um dos algoritmos.

Apesar de sempre encontrar a solução ótima, o FPT pode levar bastante tempo para ser executado com algumas instâncias. Para as utilizadas para a Figura 8, o tempo mediano do FPT foi de 3.8 s, mas o tempo médio foi de 121 s, evidenciando que para alguns casos específicos, o tempo de execução é bem alto. Para comparação, os tempos médio e mediano do PSO foram de cerca de 1.1 s, utilizando

200 partículas e 100 iterações.

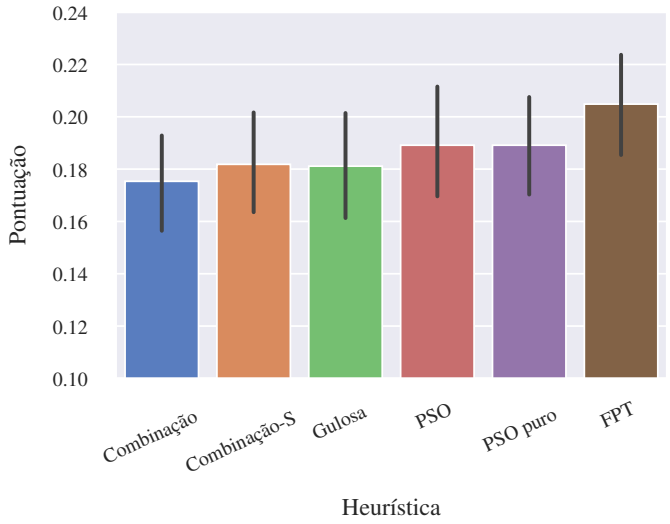


Figura 8: Comparação entre a pontuação média dos algoritmos, incluindo o FPT. Os dados incluem apenas 72 instâncias, com strings geradas utilizando $5 \leq r \leq 10$, $2 \leq s \leq 30$ e $19 \leq n \leq 36$. PSO executado com $k_E = 2$, $k_L = 0.1$, $k_G = 0.1$, 200 partículas, 100 iterações e, para as execuções incluindo posições iniciais obtidas de outras formas, $D = (1,3,3,3)$.

6 Conclusão

Observando os resultados, podemos dizer que o PSO utilizado com a representação por grafo obteve sucesso tanto em atingir soluções de qualidade para o MCSP quanto em melhorar resultados de outras heurísticas. O PSO apresentou um tempo de execução ordens de grandeza maior que os outros algoritmos, mas isso era esperado para um procedimento de busca como ele. Dentro de um cenário adequado, com um número baixo de ocorrência dos rótulos das strings, essa meta-heurística é promissora. A quantidade de testes feitos, porém, não foram suficientes para encontrar o conjunto ótimo de parâmetros para esse algoritmo nesse contexto.

Grande parte da qualidade das soluções obtidas pelo PSO pode ser atribuída à representação por grafo desenvolvida, que permite uma forma versátil de aplicar algoritmos conhecidos ao MCSP, com excelente eficiência dada pela não inclusão de blocos unitários. Trabalhos futuros podem aproveitar essa representação para aplicar outros algoritmos de otimização ao problema.

Referências

[1] Christian Blum. “Minimum common string partition: on solving large-scale problem instances”. Em: *International Transactions in Operational Research* 27.1 (2018), pp. 91–111. DOI: 10.1111/itor.12603.

[2] Christian Blum, José A. Lozano e Pinacho Davidson. "Mathematical programming strategies for solving the minimum common string partition problem". Em: *European Journal of Operational Research* 242.3 (2015), pp. 769–777. DOI: 10.1016/j.ejor.2014.10.049.

[3] Christian Blum e Günther R. Raidl. "Computational performance evaluation of two integer linear programming models for the minimum common string partition problem". Em: *Optimization Letters* 10.1 (2016), pp. 189–205. DOI: 10.1007/s11590-015-0921-4.

[4] Christian Blum et al. "Construct, Merge, Solve & Adapt: A New General Algorithm for Combinatorial Optimization". Em: *Computers & Operations Research* 68 (2016), pp. 75–88. DOI: 10.1016/j.cor.2015.10.014.

[5] Daniel Bratton e James Kennedy. "Defining a Standard for Particle Swarm Optimization". Em: *Proceedings of the IEEE Swarm Intelligence Symposium (SIS 2007)*. 2007, pp. 120–127. DOI: 10.1109/SIS.2007.368035.

[6] Laurent Bulteau e Christian Komusiewicz. "Minimum common string partition parameterized by partition size is fixed-parameter tractable". Em: *Proceedings of the 25th annual ACM-SIAM Symposium on Discrete algorithms (SODA 2014)*. SODA '14. USA: Society for Industrial e Applied Mathematics, 2014, pp. 102–121. ISBN: 978-1-61197-338-9.

[7] Laurent Bulteau et al. "A Fixed-Parameter Algorithm for Minimum Common String Partition with Few Duplications". Em: *Algorithms in Bioinformatics*. Vol. 8126. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 244–258. ISBN: 978-3-642-40453-5. DOI: 10.1007/978-3-642-40453-5_19.

[8] Xin Chen et al. "Assignment of Orthologous Genes via Genome Rearrangement". Em: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 2.4 (2005), pp. 302–315. DOI: 10.1109/TCBB.2005.48.

[9] Marek Chrobak, Petr Kolman e Jiří Sgall. "The Greedy Algorithm for the Minimum Common String Partition Problem". Em: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 84–95. ISBN: 978-3-540-27821-4. DOI: 10.1007/978-3-540-27821-4_8.

[10] Graham Cormode e S. Muthukrishnan. "The string edit distance matching problem with moves". Em: *ACM Transactions on Algorithms* 3.1 (2007), 2:1–2:19. DOI: 10.1145/1186810.1186812.

[11] S. M. Ferdous e M. Sohel Rahman. "Solving the Minimum Common String Partition Problem with the Help of Ants". Em: *Mathematics in Computer Science* 11.2 (2017), pp. 233–249. DOI: 10.1007/s11786-017-0293-5.

[12] Avraham Goldstein, Petr Kolman e Jie Zheng. "Minimum Common String Partition Problem: Hardness and Approximations". Em: *Proceedings of the 15th Annual International Symposium on Algorithms and Computation (ISAAC 2004)*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 484–495. ISBN: 978-3-540-30551-4. DOI: 10.1007/978-3-540-30551-4_43.

[13] Isaac Goldstein e Moshe Lewenstein. “Quick greedy computation for minimum common string partition”. Em: *Theoretical Computer Science* 542 (2014), pp. 98–107. DOI: 10.1016/j.tcs.2014.05.006.

[14] Dan He. “A Novel Greedy Algorithm for the Minimum Common String Partition Problem”. Em: *Bioinformatics Research and Applications*. Vol. 4463. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 441–452. ISBN: 978-3-540-72031-7. DOI: 10.1007/978-3-540-72031-7_40.

[15] Haitao Jiang et al. “Minimum common string partition revisited”. Em: *Journal of Combinatorial Optimization* 23.4 (2012), pp. 519–527. DOI: 10.1007/s10878-010-9370-2.

[16] J. Kennedy e R. Eberhart. “Particle swarm optimization”. Em: *Proceedings of the International Conference on Neural Networks (ICNN 1995)*. Vol. 4. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[17] Petr Kolman e Tomasz Waleń. “Approximating reversal distance for strings with bounded number of duplicates”. Em: *Discrete Applied Mathematics* 155.3 (2007), pp. 327–336. DOI: 10.1016/j.dam.2006.05.011.

[18] Petr Kolman e Tomasz Waleń. “Reversal Distance for Strings with Duplicates: Linear Time Approximation Using Hitting Set”. Em: *Approximation and Online Algorithms*. Vol. 4368. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 279–289. ISBN: 978-3-540-69514-1. DOI: 10.1007/11970125_22.

[19] *Handbook of Heuristics*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-07124-4. DOI: 10.1007/978-3-319-07124-4.

[20] Yuhui Shi e Russell C. Eberhart. “Parameter selection in particle swarm optimization”. Em: *Evolutionary Programming VII*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 591–600. ISBN: 978-3-540-68515-9. DOI: 10.1007/BFb0040810.

[21] Gabriel Siqueira. “Heurísticas para Problemas de Rearranjo de Genomas com Genes Multiplicados”. Dissertação de Mestrado. Brasil: Instituto de Computação, Universidade Estadual de Campinas, 2021.

[22] Gabriel Siqueira, Alexsandro Oliveira Alexandrino e Zanoni Dias. “Signed rearrangement distances considering repeated genes, intergenic regions, and indels”. Em: *Journal of Combinatorial Optimization* 46.2 (2023), p. 16. DOI: 10.1007/s10878-023-01083-w.

[23] Xin-She Yang. *Nature-inspired Metaheuristic Algorithms*. 2nd edition. Luniver Press, 2010. 148 pp. ISBN: 978-1-905986-28-6.