UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALCIDES RIBEIRO SAMPAIO NETO

ANÁLISE DE DESEMPENHO DE SOLVERS PARA O PROBLEMA DE PARTIÇÃO DE STRINGS COMUNS MÍNIMA

QUIXADÁ

2023

ALCIDES RIBEIRO SAMPAIO NETO

ANÁLISE DE DESEMPENHO DE SOLVERS PARA O PROBLEMA DE PARTIÇÃO DE STRINGS COMUNS MÍNIMA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Orientador: Prof. Dr. Paulo Henrique Macêdo de Araújo

QUIXADÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S181a Sampaio Neto, Alcides Ribeiro.
Análise de desempenho de solvers para o problema de partição de strings comuns mínima / Alcides Ribeiro Sampaio Neto. – 2023.
56 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2023.
Orientação: Prof. Dr. Paulo Henrique Macêdo de Araújo.

1. Partição de strings. 2. Programação Linear. 3. Solver. 4. Otimização. 5. Análise de Desempenho. I. Título.
CDD 004

ALCIDES RIBEIRO SAMPAIO NETO

ANÁLISE DE DESEMPENHO DE SOLVERS PARA O PROBLEMA DE PARTIÇÃO DE STRINGS COMUNS MÍNIMA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Aprovada em: __/__/_____

BANCA EXAMINADORA

Prof. Dr. Paulo Henrique Macêdo de Araújo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Wladimir Araujo Tavares
Universidade Federal do Ceará - UFC

Prof. Dr. Rennan Ferreira Dantas
Universidade Federal do Ceará - UFC

A Deus e à minha família.

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus por me dar a vida, bons pais, uma boa família e bons amigos.

Gostaria de agradecer à minha mãe Marlene Amanco da Silva por ter me incentivado a estudar diariamente.

Agradeço imensamente ao meu orientador, Paulo Henrique Macedo de Araújo. Um homem de muita virtude e de ótimos ensinamentos.

Aos meus preciosos amigos Tharlys Martins Cavalcante, Jordão Rodrigues Dantas e Sara da Silva Rocha.

De modo especial, gostaria de agradecer aos meus colegas e amigos que estiveram sempre ao meu lado na graduação, Carlos Eduardo da Silva Ferreira e Claro Henrique Sales.

A todos os citados meu mais sincero obrigado.

"Quem já conquistou o maior tesouro de ser amigo de um amigo, quem já conquistou uma mulher amável, alegre-se conosco!"

(Friedrich Schiller)

RESUMO

Neste trabalho, implementamos dois modelos de programação linear inteira (PLI) da literatura para o problema de partição de *strings* comuns mínima (PSCM) e os testamos em diferentes *solvers*, usando a técnica de *warm-start* com uma heurística gulosa, a fim de encontrar o par modelo-solver que resolva mais rápido o problema.

Palavras-chave: PSCM; Otimização; Programação Linear Inteira; OR-Tools.

ABSTRACT

In this paper, we have implemented two Integer Linear Programming (ILP) models for the Minimum Common String Partition (MCSP) problem with different solvers and using the warm-start technique, aiming to find the pair model-solver that solves the problem in the fastest way.

Keywords: MCSP; Optimization; Integer Linear Programming; OR-Tools.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	*aplication programming interface*
CLI	*command line interface*
CP	programação por restrições
MCSP	*minimum common string partition*
PL	programação linear
PLI	programação linear inteira
PLIB	programação linear inteira binária
PLIM	programação linear inteira mista
PLIP	programação linear inteira pura
PNLIM	programação não linear inteira mista
PQIM	programação quadrática inteira mista
PSCM	partição de *strings* comuns mínima

SUMÁRIO

1 INTRODUÇÃO

A comparação entre cadeias de caracteres, ou *strings*, é um problema clássico na Ciência da Computação. Esse problema possui aplicações em diversas áreas, como Biologia Computacional, Processamento de Textos e Compressão (GOLDSTEIN *et al.*, 2004).

Problemas de strings relacionados à sequência de DNA ou proteínas são abundantes na Biologia Computacional. Exemplos bem conhecidos desses problemas são: Supersequência Comum Mais Curta (GALLARDO, 2012), Cadeia de Caracteres Mais Próxima (MENESES *et al.*, 2005), Cadeia de Caracteres Mais Distante (MOUSAVI *et al.*, 2012) e Partição de Strings Comuns Mínima (PSCM). Muitos desses problemas pertencem à classe NP-Difícil e são muito custosos computacionalmente (BLUM; RAIDL, 2016).

A primeira definição do problema PSCM foi dada por Chen *et al.* (2004). Inspirado por problemas computacionais surgidos no contexto do rearranjo do genoma, como determinar se uma sequência de DNA pode ser obtida pela reordenação de subsequências de outra sequência de DNA (BLUM; RAIDL, 2016). A forma geral do problema já foi demonstrado que pertence à classe NP-difícil (GOLDSTEIN *et al.*, 2004).

O modelo proposto por (BLUM *et al.*, 2015), utilizando programação linear inteira (PLI) e uma abordagem heurística com base nesse modelo, é o atual estado-da-arte desse problema, como demonstrado pelos experimentos com problemas de *strings* de tamanho até 1000 caracteres.

Modelos de PLI podem ser solucionados utilizando *solvers* específicos. Atualmente, existem vários *solvers* disponíveis para resolver esse tipo de problema, alguns bem estabelecidos como o CPLEX (IBM, 2022) e outros mais recentes como o SCIP (SCIP TEAM, 2022). Entretanto, como observado por Anand *et al.* (2017), não existe um único *solver* que se sobressai na resolução de problemas ou nas métricas de qualidade.

Neste contexto, torna-se necessário o estudo e avaliações dos modelos do problema PSCM, para assim determinar os *solvers* e modelos mais adequados para sua solução em determinadas categorias de instâncias. Para isso, foram implementados os dois modelos descritos por Blum e Raidl (2016) e o algoritmo guloso de Chrobak *et al.* (2004). Avaliamos a performance de tempo de execução desses métodos de resolução com diferentes *solvers*.

O restante deste trabalho está organizado da seguinte maneira. No Capítulo 2 são apresentados os principais conceitos para o desenvolvimento deste trabalho, que são bioinformática, programação linear (PL), *solvers* e *OR-Tools*. No Capítulo 3 realiza-se a apresentação

e discussão dos trabalhos relacionados, com suas semelhanças e diferenças do trabalho aqui proposto. No Capítulo 4, a metodologia para o desenvolvimento deste trabalho é apresentada em detalhes, no Capítulo 5, são apresentados os resultados obtidos e no Capítulo 6 a conclusão.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os principais conceitos necessários para a execução dos experimentos propostos neste trabalho. Na seção 2.1 introduzimos o conceito de partição de *string*, explicamos formalmente o que é e como pode ser aplicado em diferentes contextos. Na seção 2.2 explicamos o que é uma partição comum de duas *strings*. Na seção 2.3 explicamos o problema abordado neste trabalho. Na seção 2.4 falamos sobre a Bioinformática. Na seção 2.5 explicamos e definimos formalmente PL e PLI. Na seção 2.6 explicamos o que são os *solvers* e abordamos os que serão usados neste trabalho. Na seção 2.7 fazemos uma introdução à plataforma OR-Tools e, por fim, na seção 2.8 explicamos o que é Heurística, bem como suas ventagens e desvantagens.

2.1 Partição de uma *string*

Uma partição de uma *string* é uma separação da *string* em partes (*substrings*). Na literatura, essas partes são chamadas de blocos. Considere a *string* $s_1 =$ "abscissa" e a partição $P_1 = \{$'a', 'b', 's', 'c', 'i', 's', 'sa'$\}$. A partição P_1 é uma partição de s_1, como podemos ver na Figura 1, onde os blocos são identificados por cores. Uma partição cujos blocos contenham apenas um caractere ou uma partição com apenas um bloco que é a própria *string* também são partições dessa *string*. O valor ou tamanho de uma partição é dado pelo número de blocos que a forma, por exemplo, $|P_1| = 7$, pois P_1 é formada por sete blocos.

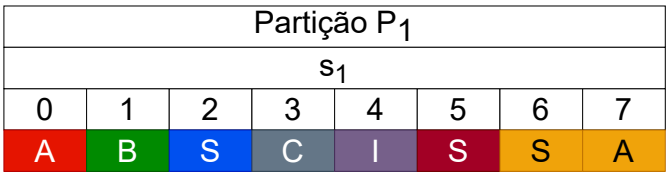
Formalmente, dada uma *string* S, uma partição de *string* é uma coleção $P = p_1, p_2, \ldots, p_n$ de *substrings* de S, onde $|P| = n$ é o número de *substrings* na partição, e a concatenação das *substrings* $p_1 \bullet p_2, \ldots, p_n$ é igual a S, onde o símbolo $\bullet$ significa uma concatenação.

Na Matemática, podemos representar uma partição de *string* como uma função $f : S \to P$, onde $f(S) = P$. Cada *substring* p_i na partição P é representada por um índice i. Portanto, uma partição de *string* é uma relação um-para-muitos entre a *string* original S e o conjunto de I que a compõem.

2.2 Partição comum de duas *strings*

Uma partição P_k é dita partição comum de duas strings s_i e s_j, se é possível concatenar todos os blocos $b_i \in P_k$, em alguma ordem, para se obter s_1 e s_2. Considere a *string* s_1 e a partição $P_1 = \{$'a', 'b', 's', 'c', 'i', 's', 'sa'$\}$ definidas na seção 2.1 e a *string* $s_2 =$ "bsasacis". Como

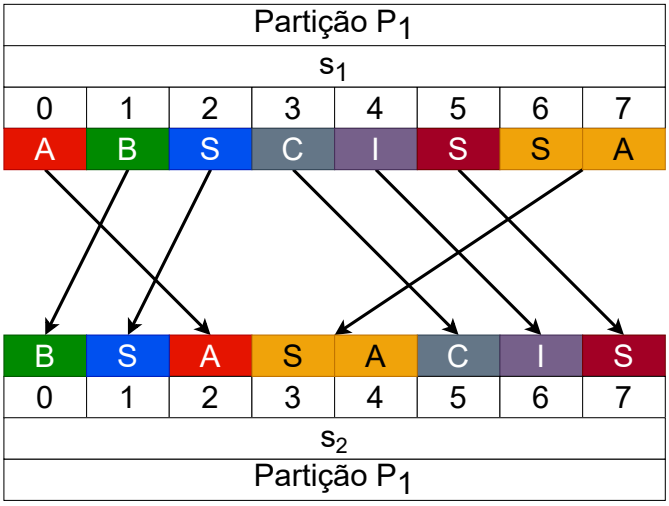
Figura 1 – Exemplo de partição P_1 na *string* s_1.



Fonte: Elaborado pelo autor.

podemos ver na Figura 2, P_1 é partição de s_2, logo P_1 é uma partição comum de s_1 e s_2.

Figura 2 – Exemplo da partição P_1 na *string* s_2.



Fonte: Elaborado pelo autor.

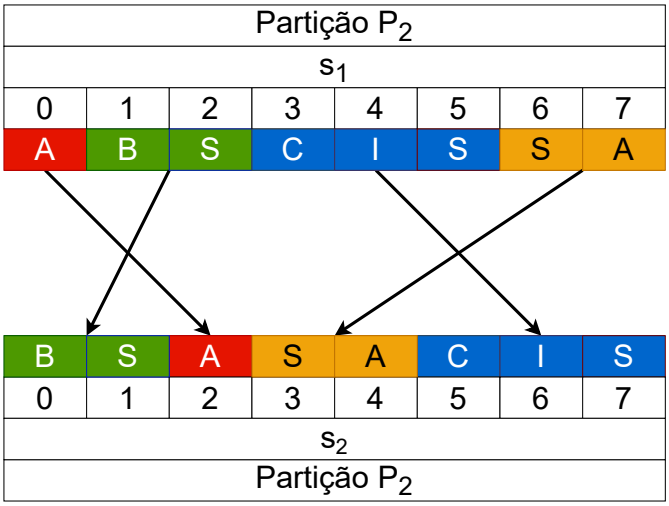
2.3 Problema da partição de *string* comum mínima

O problema de Partição de Strings Comuns Mínima (PSCM) consiste em encontrar a menor partição comum entre duas *strings*. Considere as definições das seções 2.1 e 2.2. A partição P_1 é uma solução-viável para o problema com valor igual a 7, mas não é uma solução ótima. Uma solução ótima para essa instância é a partição $P_2 = \{\text{‘a’}, \text{‘bs’}, \text{‘cis’}, \text{‘sa’}\}$, cujo valor é igual a 4, como podemos observar na Figura 3.

2.4 Bioinformática

A bioinformática é um campo de estudos que utiliza técnicas de ciência da computação, matemática e estatística para problemas da biologia para analisar e interpretar dados biológicos (LUSCOMBE *et al.*, 2001). Quando surgiu, o foco metodológico estava no desenvol-

Figura 3 – Partição comum mínima.



Fonte: Elaborado pelo autor.

vimento de algoritmos eficientes, estruturas de dados e técnicas de otimização capazes de lidar com os dados que surgiam em aplicações das ciências naturais.

Um dos atuais objetivos da bioinformática é ajudar biólogos a recolher e processar dados de genomas para estudar funções de proteínas. Outro papel importante é o de ajudar pesquisadores e companhias farmacêuticas a realizarem estudos detalhados nas estruturas das proteínas a fim de facilitar o desenvolvimento de novas drogas (COHEN, 2004).

No contexto da programação linear, a bioinformática aborda problemas como a modelagem de fenômenos biológicos, análise de dados biológicos e a extração de *insights* destes modelos e dados, bem como o foco na otimização das estruturas de dados, modelagens e da busca pelos melhores resultados de um modelo. Alguns exemplos práticos são o sequenciamento de *RNA* e *DNA*, onde técnicas de otimização de programação linear podem ser aplicadas para auxiliar na estruturação e modelagem dos dados, bem como a análise das informações e o processamento pelos melhores resultados (ALTHAUS *et al.*, 2009).

2.5 Programação linear

O termo linear, no conceito de programação linear, significa que todas as funções matemáticas dos modelos são funções lineares. Programação não se refere a programação de computadores, mas um sinônimo de planejamento. Ou seja, a PL envolve o planejamento de atividades para obter um resultado ótimo, isto é, um resultado que atinja o melhor objetivo especificado no modelo entre todos os resultados viáveis (HILLIER *et al.*, 2006).

A PL pode ser definida como o problema de maximizar uma função linear sujeita a

restrições lineares. As restrições podem ser igualdades ou desigualdades. Elas determinam a área de soluções viáveis no espaço $\mathbb{R}^n$, onde n é igual ao número de variáveis no modelo. Dentre essas soluções, as que maximizarem a função objetivo do modelo serão soluções ótimas. Programação no sentido de programação linear significa o desenvolvimento de algoritmos eficazes para a solução de problemas (MARINS, 2011).

A PL têm amplas aplicações em vários campos. Por exemplo, a PL engloba muitas das principais técnicas de soluções no campo normalmente referido como Ciência de Gestão ou Pesquisa Operacional (STRAYER, 2012).

Um modelo de PL é representado em (2.1)-(2.3), no qual n é o número de variáveis, m o número de restrições, $x \in \mathbb{R}^n$ é um vetor de variáveis de decisão, $b \in \mathbb{R}^m$ é um vetor de contantes, $F_0 : \mathbb{R}^n \to \mathbb{R}$ é a função objetivo e $F_i : \mathbb{R}^n \to \mathbb{R} \mid \forall i \in \{1,\ldots,m\}$ são as funções de restrição.

$$\text{Maximize} \quad F_0(x) \tag{2.1}$$

$$\text{Sujeito a} \quad F_i(x) \leq b_i \quad \forall i \in \{1,\ldots,m\} \tag{2.2}$$

$$x_j \geq 0 \quad \forall j \in \{1,\ldots,n\} \tag{2.3}$$

São necessários dois passos para a resolução de um problema de PL, são eles, respectivamente: a modelagem do problema, ou seja, definir matematicamente as variáveis, função objetivo e restrições, e aplicar um método para a resolução.

Existem dois métodos bem conhecidos para resolução de modelos PL: método SIMPLEX e método de pontos interiores. A complexidade de pior caso do método de pontos interiores é polinomial enquanto a do SIMPLEX é exponencial. Porém, no caso médio, o simplex possui complexidade polinomial e geralmente funciona mais rápido.

Vamos considerar um exemplo simples de programação linear. Suponha que você seja dono de uma loja de camisetas e queira determinar quantas camisetas de cada cor deve produzir para maximizar seu lucro, levando em consideração algumas restrições.

Vamos definir as variáveis de decisão que serão usadas em Equação 2.9:

- X_1: número de camisetas da cor A produzidas
- X_2: número de camisetas da cor B produzidas

Agora, vamos estabelecer as restrições:

1. Restrição de disponibilidade de material: Você só possui material para produzir no máximo 100 camisetas. Isso pode ser representado pela seguinte inequação: $X_1 + X_2 \leq 100$, definida na Equação 2.5.
2. Restrição de demanda: A demanda máxima para camisetas da cor A é de 80 unidades e para camisetas da cor B é de 120 unidades. Isso pode ser representado pelas seguintes inequações: $X_1 \leq 80$ e $X_2 \leq 120$, definidas nas equações 2.6 e 2.7.
3. Restrição de produção mínima: Você precisa produzir pelo menos 30 camisetas no total para atender às encomendas. Isso pode ser representado pela inequação: $X_1 + X_2 \geq 30$, definida na Equação 2.8.

Agora, vamos estabelecer a função objetivo:

- Função objetivo: O lucro por camiseta da cor A é de R$ 10 e da cor B é de R$ 15. Portanto, a função objetivo é maximizar o lucro total, representado por $10X_1 + 15X_2$, definidas na Equação 2.4.

Com todas as restrições e a função objetivo definidas, temos o seguinte problema de programação linear:

$$\text{Maximize } 10X_1 + 15X_2 \tag{2.4}$$

$$\text{Sujeito a } X_1 + X_2 \leq 100 \tag{2.5}$$

$$X_1 \leq 80 \tag{2.6}$$

$$X_2 \leq 120 \tag{2.7}$$

$$X_1 + X_2 \geq 30 \tag{2.8}$$

$$\text{Variáveis: } X_1 \geq 0, X_2 \geq 0 \tag{2.9}$$

Uma vez definido o problema de programação linear, podem ser aplicados algoritmos específicos, como o Simplex, para encontrar as soluções ótimas, ou seja, os valores de X_1 e X_2 que maximizam o lucro considerando todas as restrições.

2.5.1 Programação linear inteira

A programação linear inteira é um método de otimização onde ao menos uma das variáveis de decisão pertence ao conjunto dos números inteiros positivos $\mathbb{Z}_+$. Casos particulares são: programação linear inteira mista (PLIM), quando nem todas as variáveis são inteiras; programação linear inteira pura (PLIP) quando todas as variáveis são inteiras; e programação

linear inteira binária (PLIB), onde todas as variáveis são binarias, ou seja, pertencem ao conjunto $\{0,1\}$. O modelo (2.10)-(2.12) é um modelo de PLIP porque todas as variáveis são inteiras.

$$\text{Maximize} \quad F_0(x) \tag{2.10}$$

$$\text{Sujeito a} \quad F_i(x) \leq b_i \quad \forall i \in \{1,\ldots,m\} \tag{2.11}$$

$$x_j \in \mathbb{Z}_+ \quad \forall j \in \{1,\ldots,n\} \tag{2.12}$$

2.6 Solver

Optimization solvers, *linear solvers*, resolvedores ou apenas *solvers*, são programas que implementam um ou mais algoritmos para encontrar a solução-ótima de uma ou mais classes de problemas de otimização vistos na seção 2.5.

Os *solvers* estão no centro da resolução de problemas de programação linear. Com o desenvolvimento e avanço de novas tecnologias, surgem dezenas de *solvers* de programação linear capazes de resolver uma ampla gama de problemas.

Os *solvers* fornecem uma maneira melhor de lidar com as restrições conflitantes assim como as funções objetivo dos problemas lineares. Um grande número de *solvers* de otimização são desenvolvidos para resolver problemas complexos. O desenvolvimento desses *solvers* depende da natureza do problema particular a ser manipulado. Apenas um solucionador de otimização é incapaz de resolver todos os tipos de problemas da vida real (ANAND *et al.*, 2017).

O arcabouço geral dos *solvers* é composto por cinco passos principais: formulação do problema, criação do modelo, configuração do modelo, otimização e formulação da solução (ANAND *et al.*, 2017).

2.6.1 Warm-start

Warm-start é uma técnica que consiste em passar ao solver dicas para ajudá-lo a chegar na solução-inicial. As dicas são formadas por pares de variáveis e valores e podem ser uma solução-viável para o problema, mas não precisam sê-lo; podendo ser inviável ou incompleta (IBM, 2022). Dicas podem vir de problemas já resolvidos ou heurísticas, por exemplo.

2.6.2 CPLEX

O CPLEX (IBM, 2022) foi desenvolvido em C por Robert Bixby. O nome CPLEX é uma homenagem ao método simplex para solução de problemas de PL. Atualmente pertence à IBM e continua sendo desenvolvido ativamente. Suporta vários tipos de problemas de otimização, e oferece interfaces para outras linguagens além de C (ANAND *et al.*, 2017).

2.6.3 GUROBI

O GUROBI (Gurobi optmization, 2022) é um *solver* desenvolvido pela empresa Gurobi Optimizer. É um *solver* poderoso projetado do zero para rodar em *multicore* com capacidade de rodar em modo paralelo (ANAND *et al.*, 2017).

2.6.4 SCIP

O SCIP é atualmente um dos *solvers* não comerciais mais rápidos para PLIM e programação não linear inteira mista (PNLIM). E, é também uma estrutura para programação inteira por restrição e *branch-cut-and-price*. Ele permite o controle total do processo de solução e o acesso a informações detalhadas até as entranhas do solucionador (SCIP TEAM, 2022).

2.6.5 CBC

O CBC (Coin-or branch and cut) é um *solver* de programação linear inteira mista de código aberto escrito em C++. Ele pode ser usado como uma biblioteca que pode ser chamada ou usando um executável autônomo. Ele pode ser usado de várias maneiras através de vários sistemas de modelagem, pacotes, etc (FORREST *et al.*, 2022).

2.7 OR-Tools

OR-Tools é uma suíte de *softwares* para otimização gratuita e de código aberto desenvolvida pela Google para solucionar problemas de PL, PLI e programação por restrições (CP). O OR-Tools suporta várias linguagens de programação como C++, Python, Java e .Net. Além disso, ele oferece diversos *solvers*, como: CLP, CPLEX, GLOP, GLPK, Gurobi, PDLP e Xpress (GOOGLE OR-TOOLS TEAM, 2022).

2.8 Heurística

As heurísticas são projetadas para encontrar boas soluções aproximadas para problemas combinatórios difíceis que, de outra forma, não podem ser resolvidos pelos algoritmos de otimização disponíveis. Uma heurística é uma técnica de busca direta que usa regras práticas favoráveis para localizar soluções melhoradas. A vantagem das heurísticas é que elas geralmente encontram boas soluções rapidamente. A desvantagem é que a qualidade da solução em relação ao ótimo é geralmente desconhecida (TAHA, 2017).

3 TRABALHOS RELACIONADOS

Neste capítulo, são apresentados alguns trabalhos relacionados com o trabalho proposto. Por fim, na seção 3.4 é feita uma comparação entre este trabalho e os trabalhos descritos neste capítulo.

3.1 Computational Performance Evaluation of Two Integer Linear Programming Models for the Minimum Common String Partition Problem

Blum e Raidl (2016) comparam dois modelos de PLI para o problema PSCM, um modelo que usa a noção de blocos comuns e outro modelo que usa a noção de *substrings* comuns.

No primeiro modelo (apresentado em (BLUM *et al.*, 2015)), descrito com detalhes na subseção 4.1.1, os blocos estão associados às variáveis do modelo e são representados por triplas $x_i = (t_i, k_i^1, k_i^2)$, onde t_i é uma *substring* e k_i^1, k_i^2 são as posições de início da *substring* nas *strings* de entrada s_1 e s_2, respectivamente. Os blocos são então usados para estruturar as funções de restrição e a função objetivo do problema.

Foi observado pelos autores que frequentemente algumas *strings* aparecem em várias posições diferentes como *substrigs* em s_1 e/ou em s_2 e que quanto menor o alfabeto e maiores as *strings* a probabilidade é maior que algumas *substrings* menores apareçam com mais frequência.

No segundo modelo (apresentado em (BLUM; RAIDL, 2016)), e detalhado na subseção 4.1.2, as variáveis representam as ocorrências, nas *strings* de entrada, de cada *substring* em comum. Assim, sendo a variável binária $y_{t,k}^i$, $\forall t \in T$, $\forall i \in \{1,2\}$, $\forall k \in Q^i$, no qual Q é o conjunto de todas as posições iniciais onde t ocorre e i indica a *string* de entrada, caso $y_{t,k}^i = 1$, a ocorrência da *string* t na posição k na *string* s_i de entrada é escolhida para a solução, caso contrário, a ocorrência não é escolhida.

Blum e Raidl (2016) conclui então que a nova modelagem permite encontrar soluções viáveis em menos tempo e também produz melhores soluções finais na maioria dos casos em que soluções ótimas não puderam ser identificadas dentro do prazo. Uma razão importante para isso está no número de variáveis necessárias para os dois modelos. Enquanto o primeiro modelo requer variáveis $O(n^3)$ (onde n é o comprimento das *strings* de entrada), o novo modelo requer apenas variáveis $O(n^2)$.

3.2 Investigação de modelos de programação matemática para distribuição de disciplinas a professores

O trabalho de Constantino *et al.* (2021) descreve o problema de distribuição de disciplina a professores, um dos problemas mais conhecidos de alocação. O problema envolve alocar, da forma mais eficiente possível, os professores disponíveis com as disciplinas ofertadas, considerando fatores como os horários de disponibilidade dos docentes, bem como a carga horária das disciplinas. Constantino *et al.* (2021) mostra que grande parte das instituições de ensino realizam a alocação de forma manual, e que o desafio é criar um método computacional que permita automatizar esse processo de distribuição de disciplinas entre professores atendendo a um conjunto de restrições operacionais e de preferências.

Constantino *et al.* (2021) apresenta uma aplicação real do problema de distribuição de encargos didáticos a professores. São propostos três modelos matemáticos para serem aplicados ao contexto do Departamento de Informática (DIN) da Universidade Estadual de Maringá (UEM). Um modelo inicial que considera a quantidade de professores e turmas, e seus horários ao longo dos dias da semana, bem como o semestre em que uma turma é ofertada, um segundo modelo que considera a distribuição equitativa das disciplinas e, por fim, um modelo mais completo que considera outras restrições pertinentes ao departamento.

Constantino *et al.* (2021) compara os *solvers* CPLEX, GUROBI, CBC e LpSolve. O CPLEX obteve o melhor tempo de execução, porém não conseguiu solucionar para um dos parâmetros de teste na representação de ponto flutuante. O GUROBI solucionou todos os casos de teste, porém ficando atrás de CPLEX no tempo. Os outros *solvers* não obtiveram bons resultados, falhando na maioria dos casos. O GUROBI se mostrou a melhor alternativa para a solução dos modelos.

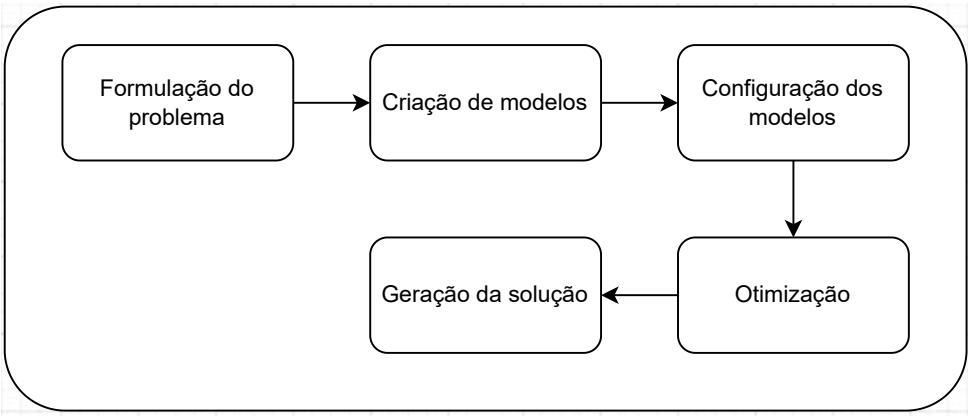
3.3 A comparative analysis of optimization solvers

O objetivo de Anand *et al.* (2017) é fazer uma análise comparativa de *solvers* de otimização. O trabalho aponta que muitos *solvers* são desenvolvidos para solucionar problemas complexos, assim, o projeto e o desenvolvimento desses *solvers* dependem da natureza do problema específico a ser tratado, e, por isso, é necessário estudar seus desempenhos.

Anand *et al.* (2017) compara três *solvers* bem conhecidos: CPLEX, GUROBI e XPRESS, da perspectiva de como conseguem lidar com várias restrições e objetivos. Para a

comparação dos *solvers* são seguidos cinco procedimentos principais: a formulação do problema, a criação de modelos, a configuração dos modelos criados, o passo de otimização e a geração da solução. A Figura 4 a seguir mostra o fluxograma destes procedimentos. Os cinco passos usados no trabalho de Anand *et al.* (2017) também são conhecidos como procedimentos gerais, por serem passos comuns da aplicação de *solvers*. Seguindo o fluxo dos procedimentos, temos que as funções objetivo e restrições são definidas na formulação do problema. Na geração do modelo, todas as funções e restrições identificadas são modeladas usando a linguagem de modelagem e as variáveis de decisão são integradas ao modelo criado. O modelo é configurado de acordo com as restrições e com as funções objetivo. As funções objetivo são otimizadas usando o modelo projetado. Então, a partir daí, as soluções otimizadas são produzidas.

Figura 4 – Fluxograma para aplicação de *solvers*.



Fonte: Adaptado de Anand *et al.* (2017).

Anand *et al.* (2017) conclui não haver um melhor solver para todas as classes de problemas ou para todas as medidas de qualidade. A adequação de um *solver* depende da natureza do problema e do tempo de computação. Mas seguindo a análise do trabalho, foi concluído que os *solvers* CPLEX e GUROBI fornecem melhores resultados com problemas da vida real. Além disso, o CPLEX é capaz de resolver problemas de programação quadrática inteira mista (PQIM). O XPRESS executa melhor que CPLEX e GUROBI em termo de escalabilidade devido as suas capacidades de *multithread*.

3.4 Este trabalho

Inspirado pelas conclusões de Anand *et al.* (2017) e pelo trabalho realizado em Constantino *et al.* (2021) este trabalho propõe implementar os dois modelos comparados em Blum e Raidl (2016) para o problema de PSCM usando a ferramenta OR-tools da Google com os *solvers* CPLEX, GUROBI, SCIP e CBC, conforme apresentado no Quadro 1. O objetivo é avaliar o desempenho de cada *solver* em cada um dos dois modelos para o problema usando um conjunto de instâncias já conhecidas na literatura e outro com novas instâncias que criamos neste trabalho.

3.5 Quadro Comparativo

Quadro 1 – Quadro comparativo dos trabalhos relacionados com este trabalho.

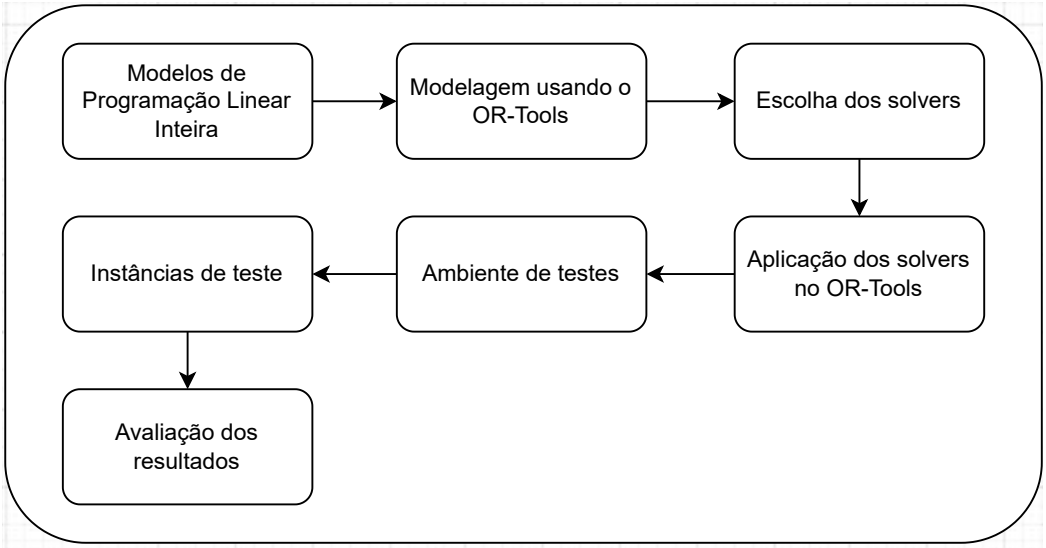
Trabalho	Problema	Solvers Usados
(BLUM; RAIDL, 2016)	MCSP	CPLEX
(ANAND *et al.*, 2017)	Vários	CPLEX, GUROBI, XPRESS
(CONSTANTINO *et al.*, 2021)	Alocação	CPLEX, GUROBI, CBC e LpSolve
Este trabalho	MCSP	CPLEX, GUROBI, SCIP e CBC

Fonte: Elaborado pelo autor.

4 PROCEDIMENTOS METODOLÓGICOS

Nesta seção, serão apresentados os procedimentos metodológicos do trabalho proposto. Um conjunto de etapas serão adotadas, tais etapas são mostradas no fluxograma da figura 5 a seguir e descritas com detalhes nas próximas seções.

Figura 5 – Fluxograma dos procedimentos metodológicos.



Fonte: Elaborado pelo autor.

4.1 Modelos de Programação Linear Inteira

Nesta seção, apresentamos detalhadamente os modelos matemáticos usados neste trabalho. O modelo proposto em (BLUM *et al.*, 2015), chamado de *Common Blocks*, e o modelo proposto em (BLUM; RAIDL, 2016), chamado de *Common Substring*. Ambos os modelos são modelos PLI.

4.1.1 Modelo Common Blocks

Para a apresentação do modelo *Common Blocks*, considere a notação a seguir:

- s_1 e s_2 são as *strings* de entrada do modelo;
- T é o conjunto de todas as *substrings* comuns entre s_1 e s_2;
- B é o conjunto dos blocos comuns;
- $n = |s_1| = |s_2|$, ou seja, é o tamanho das duas strings de entrada;

- $m = |B|$, ou seja, é a quantidade de blocos; e
- cada bloco $b_i \in B$ é uma tripla (t_i, k_i^1, k_i^2) para todo $i = 1, \ldots, m$, na qual $t_i \in T$ e k_t^1 e k_t^2 são as posições iniciais de t_i em s_1 e s_2, respectivamente.

O modelo é descrito abaixo:

$$\text{Minimize} \quad \sum_{i=1}^{m} x_i \tag{4.1}$$

$$\text{Sujeito a} \quad \sum_{i \in \{1,\ldots,m \; | k_i^1 \leq j < k_i^1 + |t_i|\}} x_i = 1 \quad \forall j = 1, \ldots, n \tag{4.2}$$

$$\sum_{i \in \{1,\ldots,m \; | k_i^2 \leq j < k_i^2 + |t_i|\}} x_i = 1 \quad \forall j = 1, \ldots, n \tag{4.3}$$

$$x_i \in \{0,1\} \qquad \forall i = 1, \ldots, m \tag{4.4}$$

A função objetivo (4.1) minimiza o número de blocos em comum. A restrição (4.2) garante que apenas um bloco seja escolhido para cada posição $j = 1, \ldots, n$ da *string* s_1 sem que haja sobreposição dos blocos escolhidos. O mesmo é efetuado para a *string* s_2 em (4.3).

4.1.2 Modelo Common Substring

Para a apresentação do modelo *Common Substring*, considere as definições a seguir:

- s_1 e s_2 são as *strings* de entrada do modelo;
- T é o conjunto de todas as *substrings* de s_1 e s_2;
- Q_t^i são os conjuntos de todas as posições entre 1 e n onde t começa nas *strings* de entrada, s_1 e s_2, respectivamente, $\forall t \in T, \forall i \in \{1,2\}$.
- $y_{t,k}^i$ representa a *substring* t iniciando na posição k da *string* s_i, $\forall t \in T, \forall k \in Q_t^i, \forall i \in \{1,2\}$.

O modelo é mostrado abaixo:

$$\text{Minimize} \quad \sum_{t \in T} \sum_{k \in Q_t^1} y_{t,k}^1 \tag{4.5}$$

$$\text{Sujeito a} \quad \sum_{t \in T} \sum_{k \in Q_t^1 | k \leq j < k + |t|} y_{t,k}^1 = 1 \quad \forall j = 1, \ldots, n \tag{4.6}$$

$$\sum_{t \in T} \sum_{k \in Q_t^2 | k \leq j < k + |t|} y_{t,k}^2 = 1 \quad \forall j = 1, \ldots, n \tag{4.7}$$

$$\sum_{k \in Q_t^1} y_{t,k}^1 = \sum_{k \in Q_t^2} y_{t,k}^2 \qquad \forall t \in T \tag{4.8}$$

$$y_{t,k}^1 \in \{0,1\}, \qquad \forall t \in T, \forall k \in Q_t^1 \tag{4.9}$$

$$y_{t,k}^2 \in \{0,1\}, \qquad \forall t \in T, \forall k \in Q_t^2 \tag{4.10}$$

A função objetivo (4.5) minimiza o número de *substrings* escolhidas. As restrições (4.6) e (4.7) garantem que para cada posição $j = 1, \ldots, n$ das *strings* de entrada s_1 e s_2, respecti-

vamente, apenas uma *substring* escolhida cobre essa posição. A restrição (4.8) garante que cada *substring* $t \in T$ é escolhida o mesmo número de vezes em s_1 e s_2.

4.2 Heurística gulosa

Em Chrobak *et al.* (2004), os autores desenvolveram um algoritmo guloso para o problema de PSCM que recebe duas *strings* relacionadas e retorna uma solução-viável para o problema. Como podemos observar no algoritmo 1, o algoritmo itera sobre as *strings* de entrada (s_1 e s_2). Em cada iteração, busca-se pela maior *substring* comum entre s_1 e s_2 que não contenha caracteres pertencentes a blocos marcados; e marcam-se os caracteres de s_1 e s_2 que formam a *substring* encontrada. Esse processo ocorre até que todos os caracteres de ambas as *strings* de entrada pertençam a blocos marcados.

Algoritmo 1: Guloso.

Entrada: Strings relacionadas S_1 e S_2
Saída: Solução-viável para o problema de PSCM
1 **while** Existirem blocos não marcados em S_1 e S_2 **do**
2 | $mcs \leftarrow$ Maior substring comum em S_1 e S_2 sem sobrepor blocos marcados
3 | Marque uma ocorrência de *mcs* em S_1 e S_2 como blocos
4 **end**
5 $(\mathbb{P}, \mathbb{Q}) \leftarrow$ sequência de blocos consecutivos em S_1 e S_2, respectivamente
6 **return** $(\mathbb{P}, \mathbb{Q})$

Fonte: Elaborado pelo autor.

4.3 Modelagem usando o OR-Tools

Nesta seção, usamos a ferramenta OR-Tools e a linguagem de programação Python para implementar os dois modelos PLI para o problema PSCM. Também apresentamos as funções auxiliares utilizadas.

4.3.1 Gerar conjunto de todas as substrings

Dada uma *string* s, a função do Código-fonte 1 retorna o conjunto de todas as *substrings* de s. O algoritmo é executado da seguinte forma: declaramos um conjunto *res* vazio; depois, no laço externo, iteramos sobre cada posição i da *string* de entrada s; no laço interno, iteramos sobre cada posição j da *string* imediatamente após a posição i, até $|s|$; acrescentamos

Código-fonte 1: Função que gera o conjunto de todas as *substrings* de uma *string*.

```
1 def gen_substrings_set(s: str) -> Set[str]:
2   res = set()
3   for i in range(len(s)):
4     for j in range(i+1,len(s)+1):
5       res.add(s[i:j])
6   return res
```

Fonte: Elaborado pelo autor.

Código-fonte 2: Função que gera o conjunto de *substrings* em comum.

```
1 def gen_common_substring_set(s1: str, s2: str) -> Set[str]:
2   subs_of_s1 = gen_substrings_set(s1)
3   subs_of_s2 = gen_substrings_set(s2)
4   return set.intersection(subs_of_s1, subs_of_s2)
```

Fonte: Elaborado pelo autor.

ao conjunto *res* a *substring* no intervalo $[i, j)$ da *string* s; ao final retornamos o conjunto *res* que contém todas as *substrings* da *string* de entrada s.

4.3.2 Gerar conjunto de substrings comuns entre duas strings

Dadas duas *strings*, s_1 e s_2, a função do Código-fonte 2, retorna o conjunto de todas as *substrings* comuns entre $s1$ e $s2$. Primeiramente, geramos o conjunto de todas as *substrings* de $s1$ e de $s2$ com a função do Código-fonte 1 e guardamos os conjuntos nas variáveis $subs\_of\_s1$ e $subs\_of\_s2$, respectivamente. Depois, retornamos a interseção desses conjuntos.

4.3.3 Gerar conjunto de blocos comuns

A função no Código-fonte 3 recebe como entrada duas *strings* s_1 e s_2 e o conjunto de todas as *substrings* comuns T e retorna o conjunto de blocos comuns, usado na implementação do modelo *Common Blocks*. O algoritmo funciona da seguinte forma: declaramos um dicionário vazio *blocks*; para cada *substring* t em T, armazenamos todas as posições iniciais em que t ocorre em ambas *strings* de entrada, $s1$ e $s2$, nas variáveis $pos\_of\_t\_in\_s1$ e $pos\_of\_t\_in\_s2$, respectivamente; após isso, iteramos sobre cada posição i em $pos\_of\_t\_in\_s1$; e para cada posição de i iteramos sobre todas as posições j a fim de gerarmos todas as combinações possíveis de blocos comuns; e, após isso, inserimos em $blocks[t]$ os pares (i, j); por fim, retornamos *blocks*.

Código-fonte 3: Algoritmo para gerar o conjunto de blocos comuns.

```
def gen_common_blocks(s1: str, s2: str, T: Set[str]) -> Dict[str, List[Tuple[int,int]]]:
  blocks: Dict[str, List[Tuple[int,int]]] = {}

  for t in T:
    pos_of_t_in_s1 = get_ocurrence_indices(s1, t)
    pos_of_t_in_s2 = get_ocurrence_indices(s2, t)

    for i in pos_of_t_in_S1:
      for j in pos_of_t_in_S2:
        if t in blocks:
          blocks[t].append((i,j))
        else:
          blocks[t] = [(i,j)]

  return blocks
```

Fonte: Elaborado pelo autor.

É importante notar a diferença entre bloco comum com bloco pertencente a uma partição. Um bloco comum (*common blocks*) é uma tripla ordenada (t_i, k_i^1, k_i^2), já um bloco pertencente a uma partição é equivalente a uma *substring*.

4.3.4 Gerar o conjunto de posições iniciais.

Dada uma *string* de entrada $s \in \{s_1, s_2\}$ e o conjunto de todas as *substrings* comuns T, a função no Código-fonte 4 retorna o conjunto de todas as posições iniciais e finais de cada *substring* $t \in T$ na *string* s. Essa função é usado no modelo *Common Substring* para gerar os conjuntos Q^1 e Q^2, no qual (4.3.6) é uma lista com dois elementos como mostrado no Código-fonte 5. O algoritmo funciona da seguinte forma: declaramos um dicionário vazio *res*; então, para cada *substring* $s[i:j]$ de s que é pertencente a T, $res[s[i:j]]$ recebe todas as posições iniciais e finais dessa *substring* em s, na forma de tuplas (i, j).

4.3.5 Implementação do Modelo Common Blocks

No Código-fonte 6, declaramos as variáveis (4.4) do modelo *Common Blocks*, representadas pelo dicionário x. Cada variável é indexada por uma tupla composta pelo índice da *substring* em T, representado por t_idx, a posição inicial em $s1$ representado por $b[0]$ e em $s2$ representado por $b[1]$. No Código-fonte 7, declaramos a restrição (4.2): para cada posição $j = 0, \ldots, N-1$ da *string* $s1$, selecionamos todos os blocos que cobrem j. Em seguida, adicionamos a restrição de que a soma dos blocos selecionados dado por *solver.Sum* tem que ser igual

Código-fonte 4: Função que gera o conjunto de posições inicias.

```
def gen_substring_pos(s: str, T: Set[str]) -> Dict[str, List[Tuple[int,int]]]:
  res: Dict[str, List[Tuple[int,int]]] = {}

  for i in range(len(s)):
    for j in range(i+1,len(s)+1):
      if s[i:j] in T:
        if s[i:j] in res:
          res[s[i:j]].append((i,j))
        else:
          res[s[i:j]] = [(i,j)]

  return res
```

Fonte: Elaborado pelo autor.

Código-fonte 5: Conjunto Q de posições iniciais.

```
Q = [gen_substring_pos(s1), gen_substring_pos(s2)]
```

Fonte: Elaborado pelo autor.

Código-fonte 6: Variáveis do modelo common blocks (4.4) no OR-Tools.

```
x = {}
for t_idx, t in enumerate(T):
  for b in B[t]:
    x[t_idx,b[0],b[1]] = solver.BoolVar(f'x[{t_idx},{b[0]},{b[1]}]')
```

Fonte: Elaborado pelo autor.

a 1. No Código-fonte 8, declaramos a restrição (4.3): para cada posição $j = 0,\ldots,N-1$ da *string* $s2$, selecionamos todos os blocos que cobrem j. Adicionamos a restrição de que a soma dos blocos selecionados dado por *solver.Sum* tem que ser igual a 1. Por fim, no Código-fonte 9 declaramos a função objetivo (4.1): selecionamos todos os blocos e chamamos *solver.Minimize* para minimizar o somatório.

4.3.6 Implementação do Modelo Common Substring

No Código-fonte 10 declaramos as variáveis (4.9), (4.10). A variável y é um dicionário, onde cada posição é indexada pelo índice da *substring* t, pelo índice do conjunto $q \in \{Q^1, Q^2\}$ e as posições iniciais e finais $k[0]$ e $k[1]$ pertencentes ao conjunto q_t. No Código-

Código-fonte 7: Restrição (4.2) no OR-Tools.

```
for j in range(N):
  blocks_at_pos_j = []
  for t_idx, t in enumerate(T):
    for b in B[t]:
      if b[0] <= j < (b[0] + len(t)):
        blocks_at_pos_j.append(x[t_idx, b[0], b[1]])

  solver.Add(solver.Sum(blocks_at_pos_j) == 1)
```

Fonte: Elaborado pelo autor.

Código-fonte 8: Restrição (4.3) no OR-Tools.

```
for j in range(N):
  blocks_at_pos_j = []
  for t_idx, t in enumerate(T):
    for b in B[t]:
      if b[1] <= j < (b[1] + len(t)):
        blocks_at_pos_j.append(x[t_idx, b[0], b[1]])

  solver.Add(solver.Sum(blocks_at_pos_j) == 1)
```

Fonte: Elaborado pelo autor.

Código-fonte 9: Função objetivo do modelo Common Blocks (4.1) no OR-Tools.

```
objetive_terms = []
for t_idx, t in enumerate(T):
  for b in B[t]:
    objetive_terms.append(x[t_idx,b[0],b[1]])

solver.Minimize(solver.Sum(objetive_terms))
```

Fonte: Elaborado pelo autor.

Código-fonte 10: Variáveis do modelo common substring (4.9), (4.10) no OR-Tools.

```
y = {}
for t_idx, t in enumerate(T):
  for q_idx, q in enumerate(Q):
    for k in q[t]:
      y[t_idx,q_idx,k[0],k[1]] = solver.BoolVar(f'y[{q_idx},{t_idx},{k[0]},{k[1]}])
```

Fonte: Elaborado pelo autor.

Código-fonte 11: Restrição (4.6) do modelo common substring no OR-Tools.

```
for j in range(N):
  substrings_at_pos_j_of_S1 = []
  for t_idx, t in enumerate(T):
    for k in Q[0][t]:
      if k[0] <= j < k[1]:
        substrings_at_pos_j_of_S1.append(y[t_idx,0,k[0],k[1]])

  solver.Add(solver.Sum(substrings_at_pos_j_of_S1) == 1)
```

Fonte: Elaborado pelo autor.

Código-fonte 12: Restrição (4.7) no OR-Tools.

```
for j in range(N):
  substrings_at_pos_j_of_S2 = []
  for t_idx, t in enumerate(T):
    for k in Q[1][t]:
      if k[0] <= j < k[1]:
        substrings_at_pos_j_of_S2.append(y[t_idx,1,k[0],k[1]])

  solver.Add(solver.Sum(substrings_at_pos_j_of_S2) == 1)
```

Fonte: Elaborado pelo autor.

fonte 11, declaramos a restrição (4.6): selecionamos todas as *substrings* que cobrem a posição j na *string* $s1$ e o somatório dessas partições deve ser igual a 1. No Código-fonte 12, declaramos a restrição (4.7): selecionamos todas as *substrings* que cobrem a posição j na *string* $s2$ e o somatório dessas partições deve ser igual a 1. No Código-fonte 13, declaramos a restrição (4.8): para cada *substring* t em T, selecionamos as partições de t em $s1$ e em $s2$ e adicionamos a restrição de que o somatório das duas partições deve ser igual. Por fim, no Código-fonte 14 declaramos a função objetivo (4.5), selecionamos todas as variáveis que representam *substrings* em $s1$ e minimizamos o somatório.

Código-fonte 13: Restrição (4.8).

```
for t_idx, t in enumerate(T):
  partitions_of_S1 = [ y[t_idx, 0, k[0], k[1]] for k in Q[0][t] ]
  partitions_of_S2 = [ y[t_idx, 1, k[0], k[1]] for k in Q[1][t] ]

  solver.Add(solver.Sum(partitions_of_S1) == solver.Sum(partitions_of_S2))
```

Fonte: Elaborado pelo autor.

Código-fonte 14: Função objetivo do modelo common substring no OR-Tools.

```
objective_terms = []
for t_idx, t in enumerate(T):
  for k in Q[0][t]:
    objective_terms.append(y[t_idx,0,k[0],k[1]])

solver.Minimize(solver.Sum(objective_terms))
```

Fonte: Elaborado pelo autor.

Código-fonte 15: Warm-start no OR-Tools.

```
solver.SetHint(variables, values)
```

Fonte: Elaborado pelo autor.

4.3.7 Warm-start com heurística gulosa

No OR-Tools, o *warm-start* pode ser passado ao *solver* usando o método *SetHint*, passando as variáveis e seus valores desejados (Código-fonte 15). Desta forma, o solver utilizará os valores como "dicas" na busca pela solução inicial e poderá acelerar o processo de busca por uma solução ótima. Para obter uma solução inicial, implementamos o algoritmo 1 no Código-fonte 16, e a solução válida retornada pelo algoritmo é repassada ao *solver* para ser usada como *warm-start*.

4.3.8 Implementação da heurística gulosa

No Código-fonte 16 implementamos algoritmo 1. O algoritmo funciona da seguinte forma: declaramos um conjunto vazio *blocks* e uma variável acumuladora *acc* que armazena o tamanho de *blocks*; enquanto o tamanho de *blocks* for menor que o tamanho da *string* $S1$,

Código-fonte 16: Implementação da heurística gulosa.

```
def chrobak(S1: str, S2: str) -> dict[str, list[tuple[int, int]]]:
    blocks = {}
    acc = 0
    while acc < len(S1):
        s, pos_in_S1, pos_in_S2 =
            longest_common_substring_without_overlapping(S1, S2, blocks)
        if s not in blocks.keys():
            blocks[s] = []
        blocks[s].append((pos_in_S1, pos_in_S2))
        acc += len(s)

    return blocks
```

Fonte: Elaborado pelo autor.

buscamos a maior *substring* comum que já não esteja em *blocks* usando uma função auxiliar, bem como sua posição inicial em $S1$ e em $S2$, e armazenamos os valores em s, pos_in_s1 e pos_in_s2 respectivamente, e acrescentamos o tamanho da *substring* s à acc; ao final retornamos um conjunto de *blocks*.

4.4 Solvers escolhidos

A modelagem foi feita usando a plataforma OR-Tools, que é independente de *solver*, de modo que, podemos usar a mesma implementação, mudando apenas o *solver*, se estiver disponível na plataforma. Assim, escolhemos usar alguns dos *solvers* voltados para problemas de PLI mais usados e que estavam disponíveis no OR-Tools, são eles: CPLEX, GUROBI, SCIP e CBC.

4.5 Uso dos solvers no OR-Tools

No OR-Tools, instanciamos um *solver* chamando o método *CreateSolver* e passando como argumento a *string* que representa o *solver* que se deseja usar, como exemplificado em Código-fonte 17. Ele retorna uma instância do *solver* recém-criada se for bem-sucedida ou *None* caso contrário. Isso pode ocorrer se a *interface* relevante não estiver vinculada ou se uma licença necessária não estiver acessível para solucionadores comerciais.

As *strings* que representam os *solvers* não diferenciam maiúsculas de minusculas e neste trabalho os seguintes nomes são usados:

- CBC_MIXED_INTEGER_PROGRAMMING ou CBC

Código-fonte 17: Criação de instância do solver CBC.

```
1 solver = pywraplp.Solver.CreateSolver('CBC')
```

Fonte: Elaborado pelo autor.

- SCIP_MIXED_INTEGER_PROGRAMMING ou SCIP
- GUROBI_MIXED_INTEGER_PROGRAMMING ou GUROBI ou GUROBI_MIP
- CPLEX_MIXED_INTEGER_PROGRAMMING ou CPLEX ou CPLEX_MIP

4.6 Ambiente de testes

Os testes foram executados em uma instância *c2-standard-8* do serviço *compute engine* da *google cloud* com oito núcleos (vCPU) e 8GB de memória com o sistema operacional Debian na versão 11 e o OR-Tools na versão 9.6. A versão dos solvers usados estão na Tabela 1.

Tabela 1: Versão dos solvers.

solver	versão
CPLEX	22.11
GUROBI	10.0.1
SCIP	803
CBC	2.10.7

Fonte: Elaborado pelo autor.

4.7 Instâncias de teste

Neste trabalho são usados um conjunto de instâncias de Ferdous e Rahman (2013) e um novo grupo de instâncias aleatórias que criamos. As instâncias de Ferdous e Rahman (2013) foram disponibilizadas pelos autores. Cada instância possui duas *strings* relacionadas, ou seja, uma é a permutação da outra.

4.8 Coleta dos dados

Os testes foram executados usando uma *command line interface* (CLI) desenvolvida para auxiliar neste trabalho[1]. A CLI permite configurar facilmente as opções de execução, como a quantidade de execuções, o *solver* a ser utilizado, e até mesmo o modelo a ser seguido. Além disso, o armazenamento dos resultados obtidos foi feito através da integração com a *aplication programming interface* (API) do Google[2], a CLI enviava os dados de cada execução para o serviço de planilhas do Google, e como forma de *backup*, os dados também eram enviados para a saída padrão, podendo ser salvos localmente na máquina que os executava, caso necessário. Instruções específicas de como usar a CLI estão no *readme file* no repositório do Github.

[1] https://github.com/srcid/tcc-tests
[2] https://cloud.google.com/apis

5 RESULTADOS

Foram testados os modelos *common blocks* e *common substring* com os *solvers* CPLEX, GUROBI, SCIP e CBC com o primeiro grupo de instâncias artificias de Ferdous e Rahman (2013) e no novo grupo de instâncias aleatórias com e sem a heurística de Chrobak *et al.* (2004) como *warm-start*. O tempo de cada execução foi obtido da média do tempo de 31 execuções. O tempo nas tabelas e figuras usadas neste capítulo estão em segundos.

A seguir, usaremos grupo 1 para nos referir ao primeiro grupo de instâncias artificias de Ferdous e Rahman (2013) e grupo 2 ao novo grupo de instâncias aleatórias geradas para este trabalho. Usaremos também as siglas "cb" e "cs" para nos referir aos modelos *common blocks* e *common substring*, respectivamente. Além disso, empregaremos o termo *speedup* para nos referirmos à diferença percentual entre duas unidades de tempo, de modo que um *speedup* positivo indica que o tempo diminuiu entre as unidades e um *speedup* negativo indica que o tempo aumentou. Nas Tabelas 2, 3 e 4 encontram-se os *speedups* obtidos ao usar *warm-start* nos *solvers* empregados. Note também que os gráficos, apesar de similares, não compartilham o eixo das ordenadas, ou seja, estão em escalas diferentes.

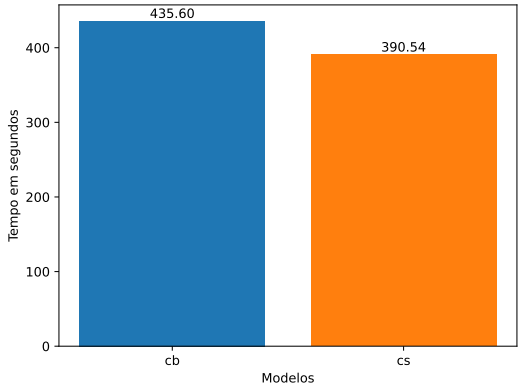
5.1 Tempo por modelo

Nesta seção são apresentados os resultados de tempo de execução entre os modelos *common blocks* e *common substrings*. O tempo de execução foi obtido somado-se o tempo de total de execução dos modelos com todos os *solvers*.

No grupo 1, o modelo *common substring* foi 11,5% mais rápido que o modelo *common blocks* sem *warm-start* e 173% com *warm-start*. Comparando os modelos com eles mesmos com e sem *warm-start*, o modelo *common substring* teve um *speedup* de 34%, já o modelo *common blocks* obteve um desempenho pior; 45% mais lento (Figuras 6 e 7).

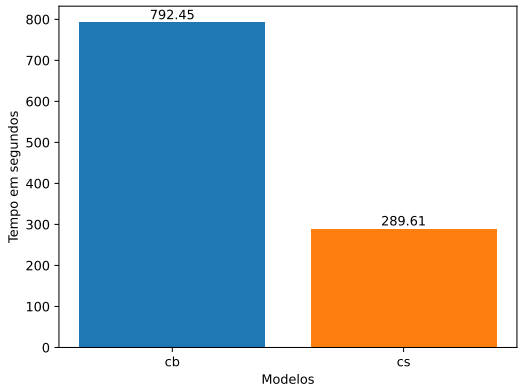
No grupo 02, o modelo *common substring* teve um *speedup*, sobre o modelo *common blocks*, de 237% sem heurística e 223% com heurística. Comparando os modelos com eles mesmos, ambos se beneficiaram do *warm-start*, o modelo *common substring* obteve um *speedup* de 20% e o modelo *common blocks* obteve um *speedup* de 25%.

Figura 6: Grupo 1 - Tempo por modelo.



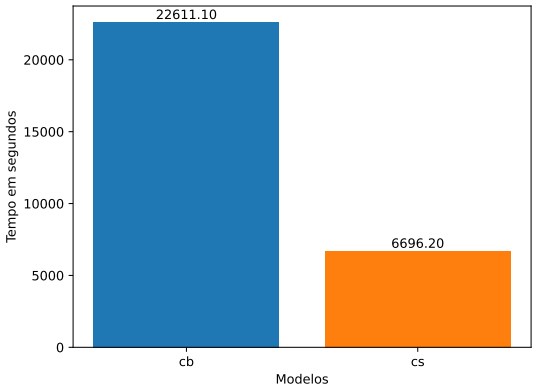
Fonte: Elaborado pelo autor.

Figura 7: Grupo 1 - Tempo por modelo com *warm-start*.



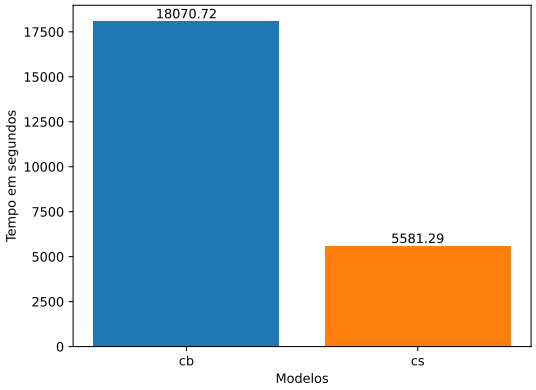
Fonte: Elaborado pelo autor.

Figura 8: Grupo 2 - Tempo por modelo.



Fonte: Elaborado pelo autor.

Figura 9: Grupo 2 - Tempo por modelo com *warm-start*.



Fonte: Elaborado pelo autor.

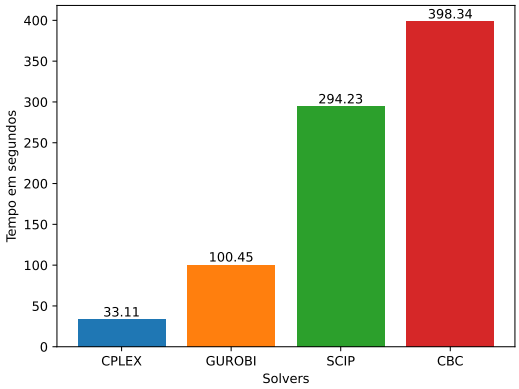
5.2 Tempo por solver com ambos os modelos

Nesta seção são apresentados os tempos de execução de cada *solver* para todas as instâncias dos grupos 1 e 2 com ambos os modelos. Nas figuras 10-13 o tempo é o somatório dos tempos de execução de todas as instâncias com ambos os modelos, *common blocks* e *common substring*.

Como podemos ver nas Figuras 10 e 11, o CPLEX foi o *solver* mais rápido no grupo 1 com e sem *warm-start*, tendo *speedup* de 203% sobre o GUROBI, 788% sobre o SCIP e 1103% sobre o CBC sem *warm-start* e tendo *speedup* de 149% sobre o GUROBI, 700% sobre o SCIP e 1804% sobre o CBC com *warm-start*.

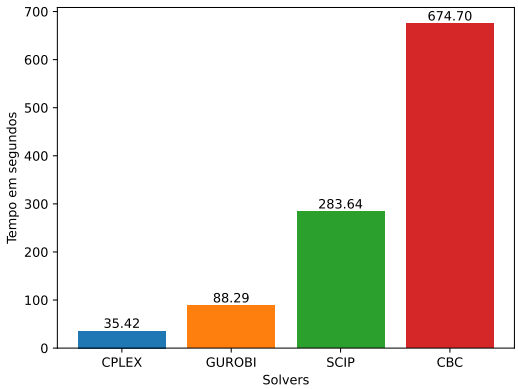
O GUROBI foi o solver mais rápido no grupo 2 com e sem *warm-up*. Com *speedup* de 1,94% sobre o CPLEX, 71,44% sobre o SCIP e 1052,11% sobre o CBC sem *warm-up* e com *speedup* de 3,05% sobre o CPLEX, 183,57% sobre o SCIP e de 794,6% sobre o CBC com

Figura 10: Grupo 1 - Tempo por solver para ambos os modelos.



Fonte: Elaborado pelo autor.

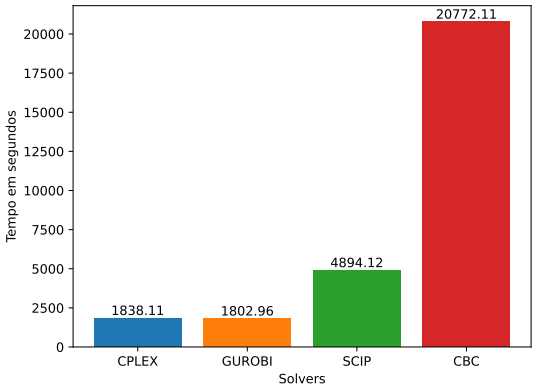
Figura 11: Grupo 1 - Tempo por solver com ambos os modelos com *warm-start*.



Fonte: Elaborado pelo autor.

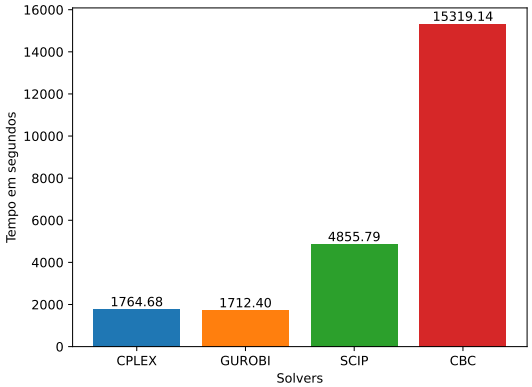
warm-up (Figuras 12 e 13).

Figura 12: Grupo 2 - Tempo por solver para ambos os modelos.



Fonte: Elaborado pelo autor.

Figura 13: Grupo 2 - Tempo por solver com ambos os modelos com *warm-start*.



Fonte: Elaborado pelo autor.

Tabela 2: *Speedup* dos *solvers* com uso de *warm-start* com ambos os modelos.

	variação	
SOLVER	GRUPO 1	GRUPO 2
CPLEX	-6,97%	3,99%
GUROBI	12,1%	5,02%
SCIP	3,59%	-0,78%
CBC	-69,37%	26,25%

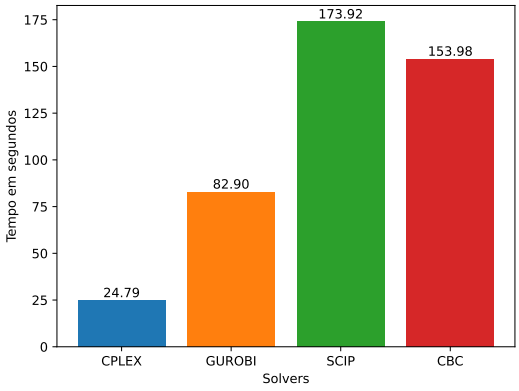
Fonte: Elaborado pelo autor.

5.3 Tempo por solver com modelo *common blocks*

Nesta seção, é apresentada a soma do tempo que cada *solver* levou para executar todas as instâncias dos grupos 1 e 2 com o modelo *common blocks*.

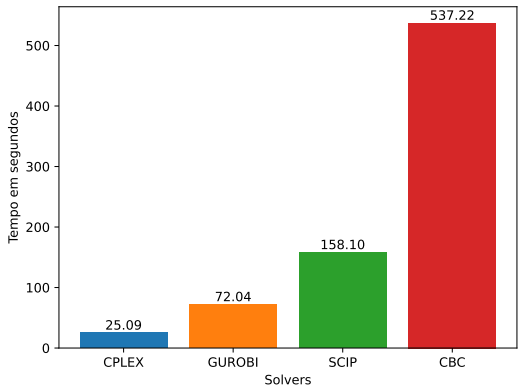
No grupo 1, o CPLEX foi o *solver* mais rápido. Com *speedup* de 234,42% sobre o GUROBI, 601,62% sobre o SCIP e 521,18%. No mesmo grupo, e com *warm-start*, o CPLEX também foi o *solver* mais rápido com *speedup* de 187,16% sobre o GUROBI, 530,17% sobre o SCIP e 2041,38% sobre o CBC (Figuras 14 e 15).

Figura 14: Grupo 1 - Tempo por *solver* modelo *common blocks*.



Fonte: Elaborado pelo autor.

Figura 15: Grupo 1 - Tempo por solver com modelo *common blocks* com *warm-start*.



Fonte: Elaborado pelo autor.

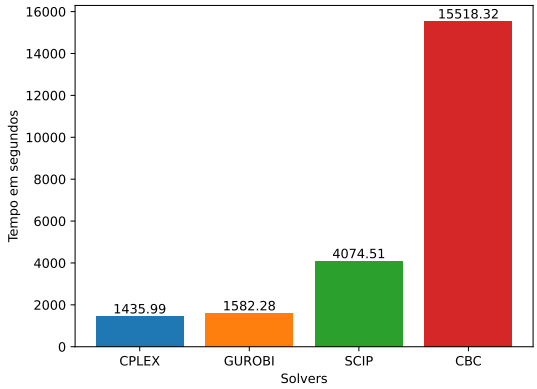
No grupo 2 sem *warm-start* o CPLEX foi o *solver* mais rápido com *speedup* de 10,19% sobre o GUROBI, 183,74% sobre o SCIP e 980,64% sobre o CBC. Com *warm-start* o GUROBI foi *solver* mais rápido com *speedup* de 9,73% sobre o CPLEX, 203,72% e 797,85% sobre o CBC (Figuras 16 e 17).

Tabela 3: *Speedup* dos *solvers* com uso de *warm-start* usando o modelo *common blocks*

	variação	
SOLVER	GRUPO 1	GRUPO 2
CPLEX	-1.21%	2.16%
GUROBI	13.10%	19.08%
SCIP	9.09%	4.55%
CBC	-248.88%	25.92%

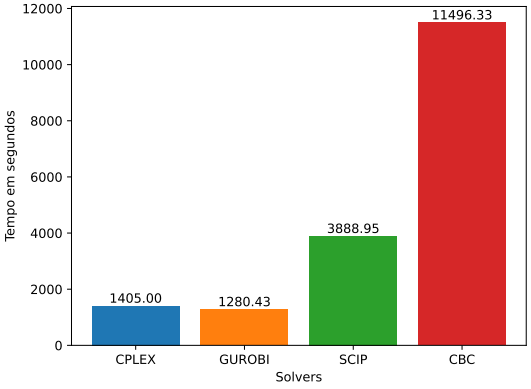
Fonte: Elaborado pelo autor.

Figura 16: Grupo 2 - Tempo por *solver* modelo *common blocks*.



Fonte: Elaborado pelo autor.

Figura 17: Grupo 2 - Tempo por solver com modelo *common blocks* com *warm-start*.



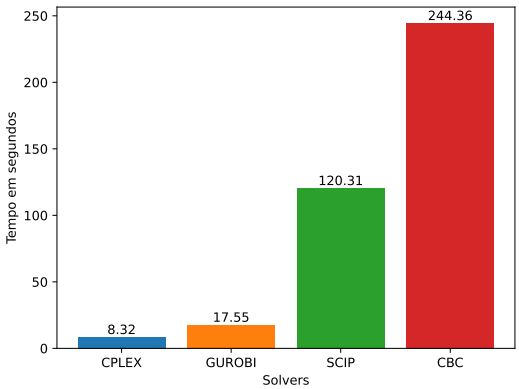
Fonte: Elaborado pelo autor.

5.4 Tempo por solver com o modelo *common substring*

Nesta seção são apresentados a soma dos tempos que cada *solver* levou para executar todas as instâncias do grupo 1 e 2 com o modelo *common substring*.

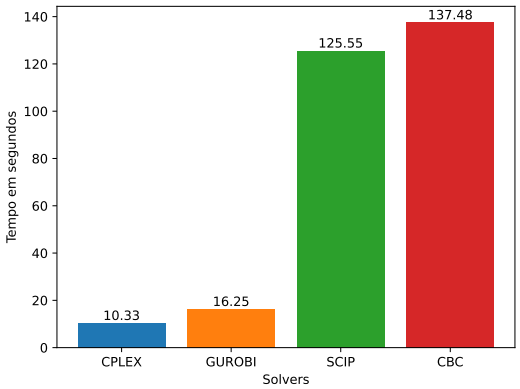
No grupo 1, o CPLEX foi o *solver* mais rápido com *speedup* de 110,92% sobre o GUROBI, 1.345,61% sobre o SCIP e 2.836,28% sobre o CBC. No mesmo grupo com o *warm-start*, o CPLEX também foi o *solver* mais rápido com um *speedup* de 187,16% sobre o GUROBI, 530,17% sobre o SCIP e 2.041,38% sobre o CBC (Figuras 18 e 19).

Figura 18: Grupo 1 - Tempo por *solver* modelo *common substring*.



Fonte: Elaborado pelo autor.

Figura 19: Grupo 1 - Tempo por solver modelo *common substring* com *warm-start*.

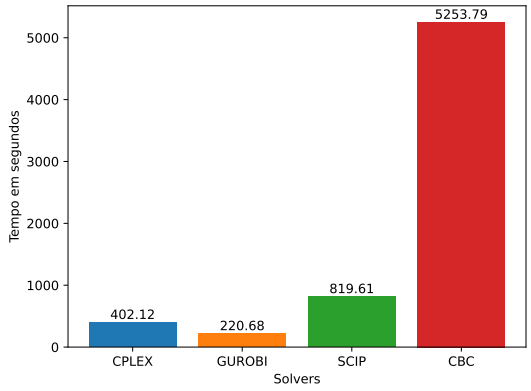


Fonte: Elaborado pelo autor.

No grupo 2, o GUROBI foi o *solver* mais rápido com um *speedup* de 82,22% sobre o CPLEX, 271,39% sobre o SCIP 2.280,71% sobre o CBC. No mesmo grupo com *warm-start* o CPLEX foi o *solver* mais rápido com um *speedup* de 20,09% sobre o GUROBI, 168,80% sobre

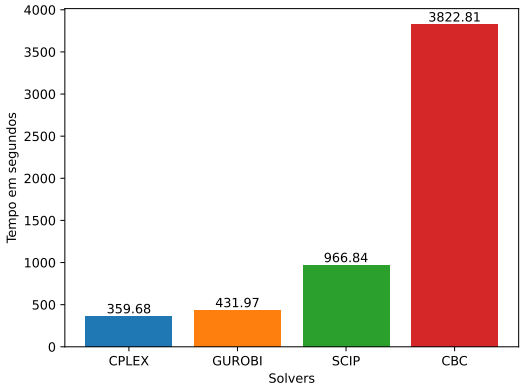
o SCIP e 962,83% sobre o CBC.

Figura 20: Grupo 2 - Tempo por *solver* modelo *common substring*.



Fonte: Elaborado pelo autor.

Figura 21: Grupo 2 - Tempo por solver modelo *common substring* com *warm-start*.



Fonte: Elaborado pelo autor.

Tabela 4: *Speedup* dos *solvers* com uso de *warm-start* usando o modelo *common substring*.

	variação	
SOLVER	GRUPO 1	GRUPO 2
CPLEX	-24.16%	10.55%
GUROBI	7.41%	-95.74%
SCIP	-4.36%	-17.96%
CBC	43.74%	27.24%

Fonte: Elaborado pelo autor.

5.5 Tempo por instância com modelo *common blocks*

Nessa seção são apresentados, nas Tabelas 5 e 6, os tempos de execução dos *solvers* por instância dos grupos 1 e 2, com e sem *warm-start*, com o modelo *common blocks*. O *solver* mais rápido para aquela instância está destacado em cinza.

No grupo 1, o GUROBI foi o *solver* mais rápido em 6 de 10 instâncias: N=113; N=115; N=118; N=119; N=123; e N=137, CPLEX foi o mais rápido nas demais instâncias: N=114; N=158; N=162; e N=170. No grupo 1 com *warm-start*, o CPLEX foi o *solver* mais rápido em 9 de 10 instâncias: N=113; N=114; N=115; N=119; N=123; N=137; N=158; N=162; e N=170. O GUROBI foi o mais rápido na instância restante: N=118 (Tabela 5).

Para o grupo 2, o GUROBI foi o *solver* mais rápido em 6 de 10 instâncias: N=180; N=190; N=200; N=220; N=235; e N=250, o CPLEX foi o mais rápido nas demais instâncias:

N=210; N=230; N=240; e N=245. No grupo 2 com *warm-start* o CPLEX foi mais rápido em 7 de 10 instâncias: N=180; N=190; N=210; N=230; N=240; N=245; e N=250, o GUROBI foi o rápido nas demais (Tabela 6).

Tabela 5: Grupo 1 - Tempos por instância com e sem *warm-start* com modelo *common blocks*.

	SEM *WARM-START*		COM *WARM-START*	
TAMANHO	*SOLVER*	TEMPO	*SOLVER*	TEMPO
113	GUROBI	0.54	CPLEX	0.15
	CPLEX	0.64	GUROBI	0.16
	CBC	0.80	CBC	2.10
	SCIP	8.37	SCIP	2.59
114	CPLEX	0.40	CPLEX	0.13
	GUROBI	0.41	GUROBI	0.18
	CBC	2.03	CBC	1.40
	SCIP	2.59	SCIP	4.41
115	GUROBI	0.77	CPLEX	0.14
	CPLEX	1.22	GUROBI	0.30
	SCIP	4.43	CBC	2.83
	CBC	5.41	SCIP	5.13
118	GUROBI	0.59	GUROBI	0.18
	CBC	0.61	CPLEX	0.24
	CPLEX	1.52	CBC	0.72
	SCIP	7.14	SCIP	2.77
119	GUROBI	0.33	CPLEX	0.13
	CPLEX	0.38	GUROBI	0.14
	SCIP	0.83	CBC	0.80
	CBC	0.87	SCIP	1.58
123	GUROBI	1.38	CPLEX	0.17
	CPLEX	1.55	GUROBI	0.44
	CBC	3.93	CBC	3.43
	SCIP	6.88	SCIP	4.16
137	GUROBI	1.32	CPLEX	0.30
	CPLEX	1.62	GUROBI	0.48
	CBC	2.36	CBC	3.89
	SCIP	17.28	SCIP	5.03
158	CPLEX	6.62	CPLEX	3.62
	GUROBI	29.37	GUROBI	6.33
	CBC	47.80	CBC	40.38
	SCIP	61.09	SCIP	61.33
162	CPLEX	8.01	CPLEX	1.63
	GUROBI	8.50	GUROBI	2.07
	SCIP	19.89	CBC	10.42
	CBC	36.16	SCIP	26.06
170	CPLEX	2.83	CPLEX	3.82
	GUROBI	39.70	GUROBI	5.97
	SCIP	45.43	SCIP	12.47
	CBC	54.01	CBC	71.49

Fonte: Elaborado pelo autor.

5.6 Tempo por instância com modelo *common substring*

Nesta seção são apresentados, nas Tabelas 7 e 8, os tempos de execução dos *solvers* com o modelo *common substring* por instância dos grupos 1 e 2. O *solver* mais performático está destacado na primeira linha de cada grupo, na cor cinza.

No grupo 1, o CPLEX foi o *solver* mais rápido em 8 de 10 instancias: N=113; N=115; N=118; N=123; N=137; N=158; N=162; e N=170, o GUROBI foi o *solver* mais rápido nas demais: N=114 e N=119. No mesmo grupo com *warm-start*, o CPLEX foi mais rápido em 9 de 10 instâncias: N=113; N=114; N=115; N=119; N=123; N=137; N=158; N=162; e N=170, e o GUROBI foi o mais rápido na instância restante: N=118 (Tabela 7).

No grupo 2, o CPLEX foi o *solver* mais rápido em 5 de 10 instâncias: N=190;

Tabela 6: Grupo 2 - Tempo por instância com e sem *warm-start* com modelo *common blocks*.

	SEM *WARM-START*		COM *WARM-START*	
TAMANHO	*SOLVER*	TEMPO	*SOLVER*	TEMPO
180	GUROBI	122.38	CPLEX	127.86
	CPLEX	162.76	GUROBI	144.91
	SCIP	478.81	CBC	533.71
	CBC	666.51	SCIP	735.66
190	GUROBI	11.56	CPLEX	6.04
	CPLEX	13.64	GUROBI	9.37
	SCIP	43.08	SCIP	55.76
	CBC	80.00	CBC	56.83
200	GUROBI	42.24	GUROBI	42.89
	CPLEX	68.67	CPLEX	49.74
	SCIP	118.19	SCIP	106.28
	CBC	326.96	CBC	132.33
210	CPLEX	4.58	CPLEX	5.02
	GUROBI	12.82	GUROBI	11.13
	SCIP	41.67	SCIP	41.89
	CBC	80.60	CBC	90.20
220	GUROBI	147.29	GUROBI	127.72
	CPLEX	475.21	CPLEX	323.84
	SCIP	494.02	SCIP	698.41
	CBC	1949.21	CBC	839.00
230	CPLEX	18.65	CPLEX	26.53
	GUROBI	29.03	GUROBI	49.47
	CBC	152.41	SCIP	88.07
	SCIP	231.90	CBC	712.66
235	GUROBI	81.25	GUROBI	34.81
	CPLEX	119.98	CPLEX	162.80
	SCIP	164.55	SCIP	283.30
	CBC	1328.64	CBC	1163.47
240	CPLEX	197.49	CPLEX	173.10
	GUROBI	345.07	GUROBI	260.12
	SCIP	967.99	SCIP	904.30
	CBC	2411.63	CBC	1482.56
245	CPLEX	236.82	CPLEX	336.15
	GUROBI	658.84	GUROBI	357.22
	SCIP	745.28	SCIP	681.47
	CBC	4416.77	CBC	1966.85
250	GUROBI	131.80	CPLEX	193.93
	CPLEX	138.20	GUROBI	242.78
	SCIP	789.01	SCIP	293.80
	CBC	4105.60	CBC	4518.72

Fonte: Elaborado pelo autor.

N=200; N=210; N=220; e N=230, e o GUROBI foi o *solver* mais rápido nas instâncias restantes: N=180; N=235; N=240; N=245; e N-N=250. No mesmo grupo com *warm-start*, o CPLEX foi *solver* mais rápido em 5 de 10 instâncias: N=190; N=210; N=220; N=230; e N=250, o GUROBI foi o *solver* mais rápido nas demais: N=180; N=200; N=235; N=240; e N=245.

5.7 Tempo por instância e modelo com melhores *solvers*

Nesta seção apresentamos o tempo dos *solvers* mais rápido por modelo nas Figuras 22-25 do grupo 1 e 2. Reforçando, na forma de gráfico, o desempenho dos *solvers* GUROBI e CPLEX, como já abordado nas Seções 5.5 e 5.6.

O modelo *common substring* foi mais rápido em todas as instâncias, com exceção da instância de tamanho N=170 do grupo 1 com *warm-start*, como podemos observar pelas Figuras 22-25.

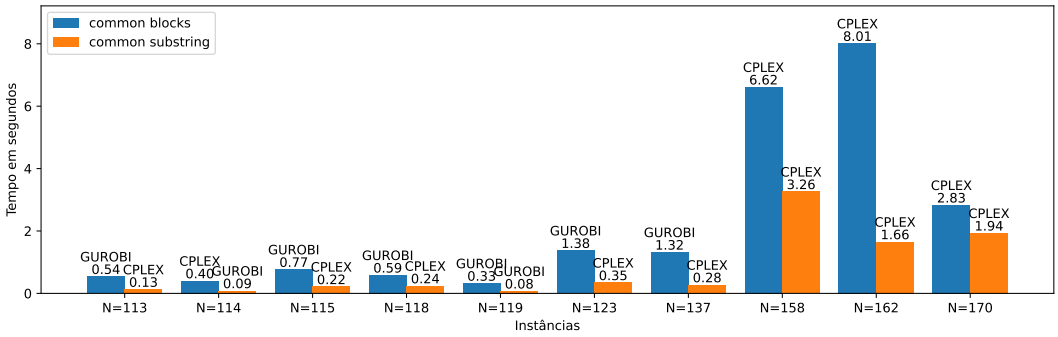
Por fim, vimos com todos os *solvers* testados, que o modelo *common susbtring* foi mais eficiente que o modelo *common blocks*. De todas as instâncias, apenas a instância de tamanho N=170 do grupo 1 com uso de *warm-start* teve desempenho melhor com o modelo

Tabela 7: Grupo 1 - Tempo por instância com modelo *common substring* com e sem *warm-start*.

	SEM *WARM-START*		COM *WARM-START*	
TAMANHO	*SOLVER*	TEMPO	*SOLVER*	TEMPO
113	CPLEX	0.13	CPLEX	0.15
	GUROBI	0.22	GUROBI	0.16
	CBC	2.53	CBC	2.10
	SCIP	5.54	SCIP	2.59
114	GUROBI	0.09	CPLEX	0.13
	CPLEX	0.12	GUROBI	0.18
	CBC	1.51	CBC	1.40
	SCIP	3.24	SCIP	4.41
115	CPLEX	0.22	CPLEX	0.14
	GUROBI	0.34	GUROBI	0.30
	CBC	1.92	CBC	2.83
	SCIP	7.82	SCIP	5.13
118	CPLEX	0.24	GUROBI	0.18
	GUROBI	0.32	CPLEX	0.24
	CBC	2.36	CBC	0.72
	SCIP	3.14	SCIP	2.77
119	GUROBI	0.08	CPLEX	0.13
	CPLEX	0.13	GUROBI	0.14
	CBC	0.71	CBC	0.80
	SCIP	1.33	SCIP	1.58
123	CPLEX	0.35	CPLEX	0.17
	GUROBI	0.54	GUROBI	0.44
	CBC	3.19	CBC	3.43
	SCIP	4.77	SCIP	4.16
137	CPLEX	0.28	CPLEX	0.30
	GUROBI	0.56	GUROBI	0.48
	CBC	2.08	CBC	3.89
	SCIP	10.38	SCIP	5.03
158	CPLEX	3.26	CPLEX	3.62
	GUROBI	5.23	GUROBI	6.33
	SCIP	34.52	CBC	40.38
	CBC	161.06	SCIP	61.33
162	CPLEX	1.66	CPLEX	1.63
	GUROBI	3.56	GUROBI	2.07
	CBC	19.40	CBC	10.42
	SCIP	32.96	SCIP	26.06
170	CPLEX	1.94	CPLEX	3.82
	GUROBI	6.62	GUROBI	5.97
	SCIP	16.60	SCIP	12.47
	CBC	49.61	CBC	71.49

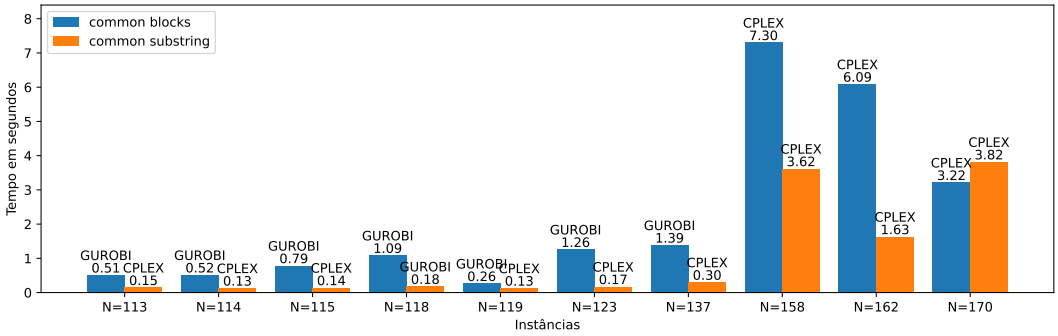
Fonte: Elaborado pelo autor.

Figura 22: Grupo 1 - Tempo por modelo com *solver* mais rápido.



Fonte: Elaborado pelo autor.

Figura 23: Grupo 1 - Tempo por modelo com *solver* mais rápido com *warm-start*.



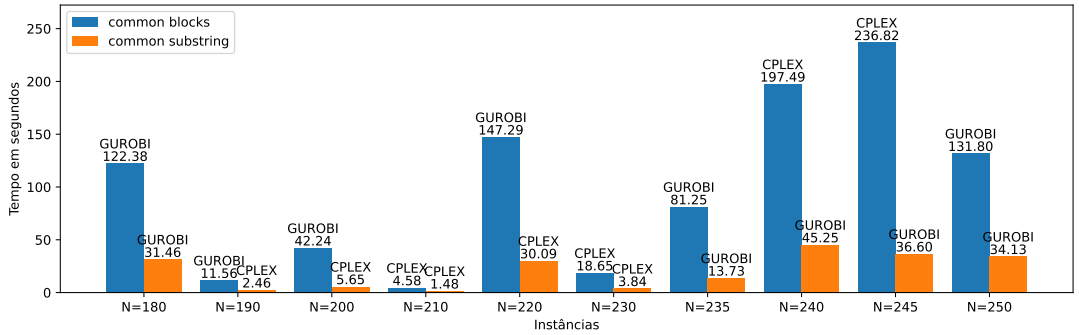
Fonte: Elaborado pelo autor.

Tabela 8: Grupo 2 - Tempo por instância com modelo *common substring* com e sem *warm-start*.

TAMANHO	SEM *WARM-START*		COM *WARM-START*	
	SOLVER	TEMPO	*SOLVER*	TEMPO
180	GUROBI	31.46	GUROBI	34.10
	CPLEX	45.80	CPLEX	39.37
	SCIP	146.75	SCIP	198.88
	CBC	1258.67	CBC	1103.15
190	CPLEX	2.46	CPLEX	2.14
	GUROBI	4.57	GUROBI	5.13
	SCIP	19.00	SCIP	23.61
	CBC	72.09	CBC	95.52
200	CPLEX	5.65	GUROBI	9.25
	GUROBI	6.76	CPLEX	19.26
	SCIP	41.16	SCIP	32.61
	CBC	100.23	CBC	76.89
210	CPLEX	1.48	CPLEX	1.50
	GUROBI	3.00	GUROBI	2.37
	CBC	8.92	CBC	8.17
	SCIP	10.33	SCIP	11.10
220	CPLEX	30.09	CPLEX	33.61
	GUROBI	37.57	GUROBI	70.75
	SCIP	123.81	SCIP	125.10
	CBC	519.08	CBC	420.26
230	CPLEX	3.84	CPLEX	6.07
	GUROBI	7.60	GUROBI	14.78
	SCIP	32.70	SCIP	34.72
	CBC	279.29	CBC	67.91
235	GUROBI	13.73	GUROBI	12.83
	SCIP	14.78	CPLEX	18.22
	CPLEX	41.28	SCIP	29.46
	CBC	99.39	CBC	117.51
240	GUROBI	45.25	GUROBI	56.90
	CPLEX	144.82	CPLEX	94.60
	SCIP	173.82	SCIP	189.78
	CBC	674.02	CBC	587.82
245	GUROBI	36.60	GUROBI	52.34
	CPLEX	50.70	CPLEX	109.60
	SCIP	135.57	SCIP	213.65
	CBC	1276.32	CBC	726.89
250	GUROBI	34.13	CPLEX	35.33
	CPLEX	76.00	SCIP	107.93
	SCIP	121.68	GUROBI	173.53
	CBC	965.78	CBC	618.70

Fonte: Elaborado pelo autor.

Figura 24: Grupo 2 - Tempo por modelo com *solver* mais rápido.



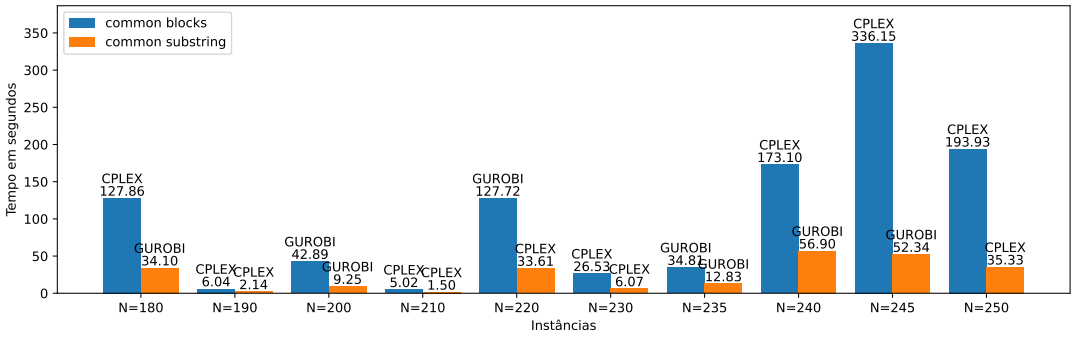
Fonte: Elaborado pelo autor.

common blocks, demonstrando que o modelo *common susbtring* consegue gerar a solução ótima de forma mais eficiente.

Com ambos os modelos, o CPLEX foi o *solver* mais rápido nos testes realizados com o grupo 1 com e sem *warm-start*. E o GUROBI foi o mais rápido no grupo 2 com e sem *warm-start*. A diferença do tempo de execução do GUROBI e do CPLEX no grupo foi pequena.

Apenas com o modelo *common blocks*, o CPLEX foi o melhor *solver*, sendo mais rápido em todos os testes, com exceção do grupo 2 com *warm-start*, no qual o GUROBI foi o mais rápido, com uma pequena diferença.

Figura 25: Grupo 2 - Tempo por modelo com *solver* mais rápido com *warm-start*.



Fonte: Elaborado pelo autor.

Apenas com o modelo *common substring*, o CPLEX foi mais rápido nos grupos 1 e 2 com *warm-start*. O GUROBI foi o mais rápido no grupo 2 sem *warm-start*, nesse caso com um *speedup* considerável em relação ao CPLEX.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, apresentamos o problema de PSCM e alguns dos modelos de PLI existentes na literatura para solucioná-lo. Falamos também dos vários *solvers* existentes para PLI e da plataforma OR-Tools que nos permite modelar problemas para vários *solvers* diferentes. Também vimos como usar a técnica de *warm-start* com heurísticas para gerar uma solução-inicial.

Com isso podemos concluir que o CPLEX e o GUROBI são os *solvers* mais rápidos para se trabalhar com o problema de PSCM e instâncias de tamanho até N=250, com ambos os modelos abordados neste trabalho. Também é possível concluir que o uso da heurística gulosa de Chrobak *et al.* (2004) como *warm-start*, apesar de agilizar na grande maioria dos casos, não garante, necessariamente, a redução no tempo de busca pela solução, já que houve casos em que os *solvers* foram mais performáticos sem o uso dessa heurística. Uma possível causa para a não redução do tempo é quando a solução que é fornecida ao *solver* não é muito boa (ou seja, está longe do ótimo global). Neste caso, ela pode levar o algoritmo de *branch-and-bound* por um caminho diferente, o que acaba aumentando o tempo geral da solução comparando ao cenário sem *warm-start*.

É interessante notar também que, apesar de não ter sido o mais eficiente entre os outros *solvers* que se destacaram neste trabalho, o SCIP foi, geralmente, mais rápido que o CBC, mantendo um *speedup* considerável sobre ele e estando abaixo apenas dos *solvers* de código-fechado e pagos.

O problema abordado neste trabalho, bem como os modelos e os *solvers* usados abrem várias direções de pesquisas. Como trabalhos futuros, dentre outros fatores que podem ser melhorados, alguns são: adicionar mais *solvers* à análise, adicionar mais heurística para serem usadas com *warm-start*, incluir mais instâncias nos experimentos e avaliar casos específicos do problema PSCM isoladamente.

REFERÊNCIAS

ALTHAUS, E.; KLAU, G.; KOHLBACHER, O.; LENHOF, H.-P.; KNUT, R. **Integer Linear Programming In Computational Biology**. In: . [S.l.: s.n.], 2009. v. 5760, p. 199–218. ISBN 978-3-642-03455-8.

ANAND, R.; AGGARWAL, D.; KUMAR, V. A comparative analysis of optimization solvers. **Journal of Statistics and Management Systems**, Taylor & Francis, v. 20, n. 4, p. 623–635, 2017. Disponível em: https://doi.org/10.1080/09720510.2017.1395182. Acesso em: 20 Set. 2022.

BLUM, C.; LOZANO, J. A.; DAVIDSON, P. Mathematical programming strategies for solving the minimum common string partition problem. **European Journal of Operational Research**, Elsevier, v. 242, n. 3, p. 769–777, 2015.

BLUM, C.; RAIDL, G. R. Computational performance evaluation of two integer linear programming models for the minimum common string partition problem. **Optimization Letters**, v. 10, n. 1, p. 189–205, Jan 2016. ISSN 1862-4480. Disponível em: https://doi.org/10.1007/s11590-015-0921-4. Acesso em: 18 Set. 2022.

CHEN, X.; ZHENG, J.; FU, Z.; NAN, P.; ZHONG, Y.; LONARDI, S.; JIANG, T. Assignment of orthologous genes via genome rearrangement. **IEEE/ACM Transactions on Computational Biology and Bioinformatics**, IEEE, v. 2, n. 4, p. 302–315, 2004.

CHROBAK, M.; KOLMAN, P.; SGALL, J. The greedy algorithm for the minimum common string partition problem. In: JANSEN, K.; KHANNA, S.; ROLIM, J. D. P.; RON, D. (Ed.). **Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 84–95. ISBN 978-3-540-27821-4.

COHEN, J. Bioinformatics—an introduction for computer scientists. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 36, n. 2, p. 122–158, jun 2004. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/1031120.1031122. Acesso em: 26 Set. 2022.

CONSTANTINO, A.; KIKUTI, D.; CLAPPIS, A. M.; BAíA, F. S. **Investigação de modelos de programação matemática para distribuição de disciplinas a professores**. In: . [s.n.], 2021. Disponível em: https://proceedings.science/sbpo/sbpo-2021/trabajos/investigacao-de-modelos-de-programacao-matematica--para-distribuicao-de-disciplinas-a-professores. Acesso em: 21 Set. 2022.

FERDOUS, S.; RAHMAN, M. S. Solving the minimum common string partition problem with the help of ants. In: SPRINGER. **International Conference in Swarm Intelligence**. [S.l.], 2013. p. 306–313.

FORREST, J.; RALPHS, T.; SANTOS, H. G.; VIGERSKE, S.; FORREST, J.; HAFER, L.; KRISTJANSSON, B.; JPFASANO; EDWINSTRAVER; LUBIN, M.; RLOUGEE; JPGONCAL1; JAN-WILLEM; GASSMANN h-i; BRITO, S.; CRISTINA; SALTZMAN, M.; TOSTTOST; PITRUS, B.; MATSUSHIMA, F.; ST to. **coin-or/Cbc: Release releases/2.10.8**. Zenodo, 2022. Disponível em: https://doi.org/10.5281/zenodo.6522795. Acesso em: 01 Out. 2022.

GALLARDO, J. E. A multilevel probabilistic beam search algorithm for the shortest common supersequence problem. **Plos one**, Public Library of Science San Francisco, USA, v. 7, n. 12, p. e52427, 2012.

GOLDSTEIN, A.; KOLMAN, P.; ZHENG, J. Minimum common string partition problem: Hardness and approximations. In: SPRINGER. **International Symposium on Algorithms and Computation**. [S.l.], 2004. p. 484–495.

GOOGLE OR-TOOLS TEAM. **OR-Tools Documentation**. 2022. Disponível em: https://developers.google.com/optimization/introduction/overview. Acesso em: 21 Set. 2022.

Gurobi optmization. **Home page**. 2022. Disponível em: https://www.gurobi.com/solutions/gurobi-optimizer/. Acesso em: 29 Nov. 2022.

HILLIER, F.; LIEBERMAN, G. J. *et al.* Introduction to operations research. **JOURNAL-OPERATIONAL RESEARCH SOCIETY**, Palgrave Macmillan Ltd, v. 57, n. 3, p. 330, 2006.

IBM. **Starting from a solution: MIP starts**. 2022. Disponível em: https://www.ibm.com/docs/en/icos/22.1.0?topic=mip-starting-from-solution-starts. Acesso em: 19 Mai. 2023.

IBM. **Visão geral**. 2022. Disponível em: https://www.ibm.com/br-pt/analytics/cplex-optimizer. Acesso em: 29 Nov. 2022.

LUSCOMBE, N. M.; GREENBAUM, D.; GERSTEIN, M. What Is Bioinfo. p. 346–358, 2001.

MARINS, F. A. S. **Introdução à pesquisa operacional**. São Paulo: Cultura Acadêmica: Universidade Estadual Paulista, 2011.

MENESES, C. N.; OLIVEIRA, C. A.; PARDALOS, P. M. Optimization techniques for string selection and comparison problems in genomics. **IEEE Engineering in Medicine and Biology Magazine**, IEEE, v. 24, n. 3, p. 81–87, 2005.

MOUSAVI, S. R.; BABAIE, M.; MONTAZERIAN, M. An improved heuristic for the far from most strings problem. **Journal of Heuristics**, Springer, v. 18, n. 2, p. 239–262, 2012.

SCIP TEAM. **SCIP Welcome**. 2022. Disponível em: https://www.scipopt.org/. Acesso em: 01 Nov. 2022.

STRAYER, J. K. **Linear Programming and Its Applications**. [S.l.]: Springer, 2012.

TAHA, H. A. **Operations research: An introduction**. 10. ed. [S.l.]: Pearson, 2017.