Instituto Federal do Norte de Minas Gerais
campus Montes Claros
Curso de Bacharelado em Ciência da Computação

COMPARAÇÃO DE HEURÍSTICAS PARA O PROBLEMA DE CORTE DE ESTOQUE UNIDIMENSIONAL INTEIRO

WELINGTON JUNIO ALVES DE SOUZA

Montes Claros
Março de 2025

WELINGTON JUNIO ALVES DE SOUZA

COMPARAÇÃO DE HEURÍSTICAS PARA O PROBLEMA DE CORTE DE ESTOQUE UNIDIMENSIONAL INTEIRO

Montes Claros
Março de 2025

Resumo

Este trabalho propõe quatro heurísticas para o problema de corte de estoque unidimensional inteiro: uma heurística gulosa, uma heurística de busca local, uma heurística de busca tabu e a utilização de algoritmo genético. O desempenho das heurísticas é comparado com métodos da literatura, incluindo heurísticas construtivas, residuais e algoritmos genéticos. A otimização do corte de materiais, com foco na minimização de perdas e custos, é um desafio significativo em várias indústrias. Os resultados computacionais mostram que as heurísticas de algoritmo genético juntamente com busca tabu se destacam como as abordagens mais eficazes, alcançando os melhores resultados em diversas instâncias. A heurística gulosa, embora adequada para instâncias simples, apresentou desempenho inferior quando comparada com as outras heurísticas, enquanto a busca local apresentou resultados equilibrados. Nos cenários mais complexos, os algoritmos genéticos demonstraram superioridade, exibindo maior robustez e capacidade de exploração do espaço de soluções. Esses achados sugerem que, embora todas as heurísticas propostas apresentem desempenho competitivo, abordagens mais avançadas, como os algoritmos genéticos, são mais adequadas para lidar com instâncias mais desafiadoras.

Abstract

This work proposes four heuristics for the integer one-dimensional stock cutting problem: a greedy heuristic, a local search heuristic, a tabu search heuristic and the use of a genetic algorithm. The performance of the heuristics is compared with methods from the literature, including constructive heuristics, residuals and genetic algorithms. Optimizing material cutting, with a focus on minimizing losses and costs, is a significant challenge in several industries. The computational results show that genetic algorithm heuristics together with tabu search stand out as the most effective approaches, achieving the best results in several instances. The greedy heuristic, although suitable for simple instances, presented lower performance when compared to the other heuristics, while the local search presented balanced results. In the most complex scenarios, genetic algorithms demonstrated superiority, exhibiting greater robustness and ability to explore the solution space. These findings suggest that, although all proposed heuristics present competitive performance, more advanced approaches, such as genetic algorithms, are better suited to dealing with more challenging instances.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O problema de corte de estoque unidimensional inteiro é amplamente estudado na pesquisa operacional devido à sua relevância prática na redução de desperdício de materiais. Esse problema ocorre em indústrias como metalurgia, papel e celulose, móveis e outras, onde é necessário cortar matérias-primas em tamanhos menores para atender a demandas específicas, minimizando perdas e maximizando o aproveitamento dos recursos disponíveis.

As primeiras formulações desse problema remontam à década de 1960, com os trabalhos pioneiros de Gilmore e Gomory, que propuseram abordagens baseadas em programação linear para minimizar custos associados ao atendimento de demandas específicas Gilmore & Gomory [1961]. A complexidade do problema reside na identificação do conjunto ideal de cortes que atenda à demanda com o menor desperdício, tornando-o desafiador tanto em termos computacionais quanto práticos.

Diferentes abordagens têm sido propostas na literatura para resolver esse problema. Métodos exatos, como programação linear e algoritmos de branch and bound, oferecem soluções ótimas, mas são computacionalmente inviáveis para instâncias de grande porte. Por outro lado, heurísticas e meta-heurísticas, como algoritmos gulosos, genéticos e busca tabu, destacam-se por fornecerem soluções aproximadas em tempo reduzido, sendo mais adequadas para aplicações práticas.

Neste estudo, propomos o desenvolvimento e a comparação de diferentes heurísticas para o problema de corte de estoque unidimensional inteiro. Inicialmente, implementamos uma heurística gulosa, que busca uma solução inicial rápida e eficiente. Em seguida, introduzimos uma heurística de busca local, explorando quatro vizinhanças: troca de itens e reorganização de itens, avaliadas tanto pela estratégia de primeira melhora quanto pela de melhor melhora, desenvolvemos também uma heurística de busca tabu, aplicada às mesmas vizinhanças, para ampliar o espaço de busca e evitar soluções repetidas ou subótimas, e, por fim, o uso de algoritmos genéticos para termos

a comparação das soluções mais ótimas conhecidas.

Os melhores resultados dessas heurísticas serão comparados com três estudos relacionados: Poldi & Arenales [2003], Boleta et al. [2005] e Heis et al. [2006], que empregaram abordagens construtivas, residuais e genéticas. A análise comparativa incluirá o desempenho computacional e a eficiência das soluções na minimização do desperdício de materiais, proporcionando uma visão abrangente das vantagens e limitações de cada método.

1.1 Organização do Documento

Após esta introdução, o documento está organizado da seguinte forma. O Capítulo 2 apresenta a definição formal do problema, abordando os principais conceitos e formulações utilizadas na literatura. No Capítulo 3, é realizada uma revisão bibliográfica detalhada, com foco nos estudos relacionados utilizados para comparação. O Capítulo 4 detalha as heurísticas propostas, incluindo a descrição das estratégias gulosa, de busca local, de busca tabu e algoritmo genético, além de suas respectivas implementações. No Capítulo 5, são apresentados os resultados experimentais, com análise comparativa entre os métodos heurísticos e meta-heurísticos. Finalmente, o Capítulo 6 traz as conclusões e as contribuições deste trabalho, apontando também para possíveis direções futuras.

Capítulo 2

Definição do Problema

O prroblema de corte de estoque unidimensional inteiro consiste em determinar a melhor maneira de cortar unidades de material bruto, como barras ou bobinas, em partes menores, atendendo a uma demanda específica de itens com comprimentos variados. O objetivo principal é minimizar o desperdício de material, ou seja, as sobras resultantes dos cortes que não podem ser reutilizadas para satisfazer a demanda. Este problema tem alta relevância em indústrias onde a eficiência no uso do material impacta diretamente os custos de produção.[Gilmore & Gomory [1961]].

2.1 O Problema

2.1.1 Formulação do Problema

Formalmente, o problema pode ser descrito da seguinte forma:

1. **Entradas:**
   - **Conjunto de Barras:** Um estoque de K tipos de barras, cada uma com comprimento L_k (onde $k = 1, 2, \ldots, K$) e uma quantidade limitada de unidades e_k.
   - **Demanda de Itens:** Um conjunto de m itens, cada um com comprimento l_i e demanda d_i (onde $i = 1, 2, \ldots, m$).
2. **Objetivo:** Determinar um conjunto de cortes que minimize o desperdício total de material, isto é, a soma dos comprimentos das sobras que não podem ser utilizadas para atender à demanda, garantindo que a produção seja suficiente para atender a todas as demandas especificadas.
3. **Restrições e Variantes:**

- A produção deve atender exatamente à demanda, sem excedentes ou cortes desnecessários.
- Em algumas configurações, múltiplos padrões de corte podem ser utilizados, permitindo a otimização do aproveitamento por meio de diferentes combinações para cada tipo de barra disponível.

2.1.2 Importância e Aplicações

O prroblema de corte de estoque unidimensional inteiro possui aplicações práticas em setores como metalurgia, papel, madeira e móveis, nos quais materiais de comprimento fixo precisam ser cortados para atender a demandas específicas. A eficiência nesse processo reduz não apenas os custos associados à matéria-prima, mas também promove práticas sustentáveis, alinhadas às exigências de produção moderna.

Além disso, a otimização no corte de materiais aumenta a competitividade industrial, particularmente em cenários de produção sob demanda ou em indústrias onde a flexibilidade no atendimento ao cliente é crucial. A capacidade de minimizar desperdícios e atender demandas com eficiência resulta em processos mais lucrativos e com menor impacto ambiental.

2.1.3 Complexidade do Problema

O problema de corte de estoque unidimensional inteiro é classificado como *NP-árduo*, indicando que não existe uma solução eficiente conhecida que resolva todas as instâncias em tempo polinomial[Boleta et al. [2005]]. Sua complexidade advém da necessidade de explorar diversas combinações de cortes, que crescem exponencialmente com o aumento do número de itens e barras disponíveis.

Embora métodos exatos, como programação linear e algoritmos de *branch and bound*, possam garantir a solução ótima, eles se tornam inviáveis para instâncias de grande porte devido ao alto custo computacional. Por esse motivo, heurísticas e metaheurísticas são amplamente empregadas, oferecendo soluções aproximadas em tempos computacionais aceitáveis. Entre essas abordagens destacam-se algoritmos gulosos, genéticos, busca tabu e heurísticas construtivas.

2.1.4 Motivações para o Estudo

O estudo e a otimização desse problema são motivados por vários fatores:

- **Desafios Computacionais:** A busca por estratégias mais eficientes é essencial para lidar com a crescente complexidade de instâncias reais do problema.

- **Redução de Custos:** A otimização no corte de materiais pode gerar economias significativas em indústrias onde o custo da matéria-prima é um fator relevante.

- **Avanços em Algoritmos:** O desenvolvimento de heurísticas inovadoras tem permitido obter soluções de qualidade em menor tempo.

- **Sustentabilidade:** Reduzir desperdícios contribui para práticas industriais mais sustentáveis e alinhadas às demandas atuais por processos produtivos ambientalmente responsáveis.

2.2 Gerador Aleatório

Para realizar testes e comparações das heurísticas desenvolvidas, foi necessário criar um gerador aleatório de instâncias que simulasse o problema de corte de estoque unidimensional inteiro. Esse gerador foi baseado no modelo proposto por Poldi & Arenales [2003], seguindo a estrutura definida por Gau & Wäscher [1995], e permite a construção de instâncias com diversas configurações de barras, itens e demandas.

O objetivo principal do gerador é proporcionar um conjunto amplo e variado de cenários, desde problemas mais simples, com poucos tipos de barras e itens, até configurações mais complexas, envolvendo uma maior diversidade de comprimentos e quantidades. Essa diversidade é essencial para avaliar a robustez e eficiência da heurística proposta, além de possibilitar comparações com resultados de outros estudos.

As instâncias geradas seguem as configurações descritas a seguir:

1. **Número de Tipos de Barras (K)**: Representa a quantidade de diferentes comprimentos de barras disponíveis em estoque. O valor de K é definido aleatoriamente como 3, 5 ou 7, representando estoques pequenos, médios e grandes.

2. **Comprimento das Barras**: O comprimento de cada barra é sorteado dentro do intervalo de 10 a 100 unidades, garantindo uma variedade significativa de tamanhos. Cada tipo de barra possui um comprimento fixo e uma quantidade limitada em estoque.

3. **Número de Tipos de Itens (m)**: Define a diversidade de itens demandados, variando entre 5, 10, 20 e 40 tipos. Essa variação busca simular demandas simples e complexas, refletindo cenários práticos encontrados em diferentes indústrias.

4. **Comprimento dos Itens**: Os itens são categorizados com base no comprimento médio das barras disponíveis:

- **Itens Pequenos (P)**: Comprimentos entre 1% e 20% do comprimento médio das barras.
- **Itens Médios (M)**: Comprimentos entre 1% e 80% do comprimento médio das barras.

Essa categorização permite testar a heurística em cenários com itens menores e maiores em relação às barras disponíveis.

5. **Demanda de Itens**: Cada tipo de item possui uma demanda gerada aleatoriamente entre 1 e 10 unidades, simulando variações práticas no consumo.

6. **Quantidade de Barras em Estoque**: A disponibilidade de cada tipo de barra é sorteada dentro do intervalo de 1 até metade do número total de tipos de itens (m), limitando o estoque e adicionando um aspecto realista ao problema.

Com essas configurações, o gerador aleatório produz instâncias que refletem cenários reais, permitindo avaliar a eficiência da heurística em uma ampla gama de problemas.

O pseudocódigo do gerador aleatório é apresentado na Figura 2.1, detalhando o processo de construção das instâncias. A implementação foi realizada na plataforma Google Colab [Google [2023]], utilizando o ambiente de desenvolvimento Python [Python [2023]]. O Google Colab oferece recursos computacionais gratuitos, permitindo a execução de algoritmos. Neste estudo, utilizamos uma instância padrão equipada com processador Intel Xeon de 2.20 GHz, 12 GB de memória RAM e sistema operacional Linux Ubuntu 18.04, utilizando a biblioteca `random` para sorteio dos valores. Essa configuração garante reprodutibilidade e flexibilidade para ajustar os parâmetros conforme necessário.

2.2.1 Motivações para o Uso do Gerador Aleatório

A utilização do gerador aleatório de instâncias permite uma análise robusta e realista das heurísticas, pois cria cenários diversos e complexos que representam a variabilidade encontrada em aplicações reais. Cada combinação de K, m, e tipo de item (P ou M) gera uma classe de instâncias, e para cada classe são criadas 20 instâncias independentes, garantindo uma quantidade de dados suficiente para avaliar o desempenho e a eficiência das heurísticas propostas.

Além disso, o gerador possibilita a criação de instâncias de corte que variam em complexidade, desde casos simples com poucos tipos de barras e itens, até cenários mais complexos com uma variedade maior de comprimentos e demandas. Esse grau de flexibilidade é essencial para testar a escalabilidade das heurísticas e identificar

```
FUNÇÃO gerar_instancia(K, m, tipo_tamanho):
    // Gerar os comprimentos das barras no estoque
    Para i de 1 até K faça:
        comprimento_barras[i] = número aleatório entre 10 e 100
    // Gerar a disponibilidade das barras no estoque
    Para i de 1 até K faça:
        disponibilidade_barras[i] = número aleatório entre 1 e (100 * m / 2)
    // Calcular o comprimento médio das barras
    comprimento_medio = soma de todos os elementos em comprimento_barras dividido por K
    // Definir os limites para o tamanho dos itens
    Se tipo_tamanho é "P" então:
        v2 = 0.2  // Limite superior para itens pequenos
    Senão:
        v2 = 0.8  // Limite superior para itens médios
    v1 = 0.01  // Limite inferior para o comprimento dos itens
    // Gerar os comprimentos dos itens
    Para i de 1 até m faça:
        comprimento_item = comprimento_medio * número aleatório entre v1 e v2
        comprimentos_itens[i] = arredondar(comprimento_item para inteiro)
    // Gerar a demanda para cada tipo de item
    Para i de 1 até m faça:
        demandas[i] = número aleatório entre 1 e 10
    // Retornar a instância gerada
    Retornar um dicionário contendo:
        'K': K,
        'comprimentos_barras': comprimento_barras,
        'disponibilidade_barras': disponibilidade_barras,
        'm': m,
        'comprimentos_itens': comprimentos_itens,
        'demandas': demandas,
        'tipo_tamanho': tipo_tamanho
```

Figura 2.1. Pseudocódigo do gerador aleatório de barras para corte e suas demandas proposto para este estudo. Fonte: Autória Própria.

as configurações que proporcionam os melhores resultados em termos de redução de desperdício de material.

Em resumo, o gerador aleatório simula com precisão as condições industriais, permitindo uma análise detalhada e uma avaliação crítica do desempenho das heurísticas aplicadas ao problema de corte de estoque.

Capítulo 3

Revisão Bibliográfica

A otimização do corte de estoque unidimensional tem sido amplamente estudada na literatura, dada sua importância em diversos setores industriais. Ao longo dos anos, diferentes abordagens e técnicas foram propostas para resolver este problema, variando desde métodos heurísticos até meta-heurísticos. Neste contexto, investigações como as conduzidas por Gramani [2008], Santiago [2016], Heis et al. [2006], Boleta et al. [2005] e Poldi & Arenales [2003] revelam uma diversidade de soluções, cada uma adaptada a suas restrições do problema. Alguns trabalhos focam em abordagens tradicionais de programação linear, enquanto outros exploram métodos mais modernos, como algoritmos genéticos, busca tabu e outras heurísticas.

O artigo de Gramani [2008] explora a dinâmica do processo de corte em sistemas produtivos, onde a gestão eficiente dos custos é crucial. A autora destaca que a literatura tradicionalmente foca em dois objetivos principais: a minimização da perda de material durante o corte e a redução da troca de padrões de corte. Enquanto a minimização das perdas tem sido amplamente estudada, especialmente em indústrias que lidam com matéria-prima cara, a minimização da troca de padrões é um desafio menos abordado, frequentemente devido à sua natureza *NP-difícil*.

Gramani [2008] propõe um método que divide o problema de corte em subproblemas menores, tornando a abordagem mais gerenciável e eficiente. Essa estratégia permite uma análise mais detalhada dos *trade-offs* entre os dois objetivos. A autora realiza experimentos computacionais para avaliar a eficácia de seu método, demonstrando que, em muitos casos, equilibrar a minimização das perdas e a troca de padrões resulta em soluções mais vantajosas do que a abordagem focada em um único objetivo.

Os resultados indicam que a aplicação do método proposto pode reduzir significativamente os custos totais, mantendo a eficiência operacional. A análise computacional revela que, ao considerar simultaneamente os custos de perda e de troca de padrões, as indústrias podem melhorar o desempenho, otimizando não apenas a utilização de

materiais, mas também o tempo e os recursos no processo de corte. Assim, o trabalho de Gramani [2008] contribui para a literatura sobre corte de estoque, oferecendo uma nova perspectiva sobre a tomada de decisão em ambientes produtivos e destacando a importância de uma abordagem equilibrada na busca por eficiência.

O estudo de Santiago [2016] aplica o método simplex com geração de colunas para resolver problemas de programação linear associados ao corte de estoque unidimensional. A abordagem inicial resolve o problema relaxado para obter uma solução fracionária, posteriormente ajustada para uma solução inteira utilizando heurísticas residuais, com destaque para a Residual Nova, proposta por Poldi & Arenales [2003].

O trabalho apresenta dois casos de aplicação. No primeiro, analisa o corte de objetos de comprimento $L = 194$, considerando três tipos de itens com comprimentos específicos e demandas associadas. A solução inicial, obtida pelo método simplex, gera uma matriz de padrões de corte e um vetor solução fracionário, posteriormente ajustado para atender às demandas. No segundo caso, aborda a produção de estruturas de gaiolas metálicas, utilizando barras de 300 cm para atender a uma demanda diversificada. A combinação do método simplex com a heurística residual permite alcançar uma solução inteira eficiente, maximizando o uso das barras e minimizando desperdícios.

Os resultados demonstram a eficácia da abordagem proposta. No primeiro caso, a solução inicial foi ajustada para atender à demanda de forma eficiente, enquanto no segundo, a heurística residual gerou soluções inteiras práticas e otimizadas. A matriz de padrões de corte e o vetor solução final para o segundo caso evidenciam o potencial do modelo para aplicações industriais, contribuindo para a redução de custos e a melhoria de processos produtivos.

O artigo de Heis et al. [2006] apresenta uma solução para o problema de corte de estoque. A solução proposta utiliza algoritmos genéticos, uma abordagem de programação evolutiva que incorpora conceitos como seleção, reprodução, mutação e avaliação, diferente das técnicas tradicionais.

O método inicia com a geração de uma população inicial de soluções viáveis, representadas como cromossomos, cujos genes correspondem aos padrões de corte. A partir dessa base, ocorre um processo iterativo de evolução, onde as soluções mais promissoras têm maior probabilidade de gerar novas combinações. Uma heurística construtiva é utilizada para garantir a factibilidade das soluções ao longo das gerações. A cada iteração, as soluções são avaliadas com base na perda de material, e as menos eficientes são substituídas por soluções de maior qualidade, até atingir um número predefinido de gerações.

Os testes computacionais implementados basearam-se no modelo de Gau & Wäscher [1995], avaliando 240 instâncias divididas em 12 classes, variando o número de barras,

itens e tamanhos. Os resultados mostram que o algoritmo genético proposto supera significativamente métodos tradicionais, como os de Poldi & Arenales [2003], tanto na redução da perda de material quanto na diminuição do número de padrões de corte. Por exemplo, na classe C1 (3 barras e 5 itens de tamanho pequeno), a heurística alcançou uma perda média de 105,85, enquanto os métodos comparados apresentaram perdas médias entre 152,40 e 166,85. Na classe C11 (7 barras e 20 itens de tamanho pequeno), a perda média foi reduzida para 16,50, destacando a eficiência do método.

Além da redução de perdas, o algoritmo demonstrou capacidade de gerar um menor número de padrões de corte, aspecto crítico para aplicações práticas, pois mudanças frequentes de padrões aumentam custos e tempos de configuração. Na classe C10, por exemplo, o número médio de padrões foi reduzido para 6,95, enquanto métodos tradicionais apresentaram valores entre 8,55 e 9,45.

O estudo de Boleta et al. [2005] investiga o Problema de Corte de Estoque Unidimensional Inteiro, propondo uma heurística inovadora baseada em conceitos de Algoritmos Genéticos para enfrentar os desafios combinatórios e de integralidade dessa classe de problemas. A abordagem utiliza seleção, reprodução e mutação, onde cada solução é representada como um indivíduo, com genes correspondendo aos padrões de corte. O valor utilitário do indivíduo é calculado com base na perda total e no número de padrões gerados, permitindo priorizar entre minimizar desperdícios ou reduzir o número de padrões.

Nos testes computacionais, baseados no método de Gau & Wäscher [1995], a heurística foi avaliada quanto à perda total, número de padrões de corte e barras utilizadas. Os resultados destacaram a eficiência da abordagem proposta, que superou métodos clássicos e soluções previamente desenvolvidas pelos autores em dois dos três critérios analisados, especialmente na redução do número de padrões—ainda mais relevante em aplicações práticas por reduzir setups e aumentar a produtividade.

Apesar da eficácia, a estabilidade dos resultados foi um desafio em alguns casos, com métodos clássicos, como o B&B, apresentando maior consistência na minimização de perdas. Contudo, a nova heurística demonstrou-se mais eficiente no geral, superando inclusive o método de Poldi & Arenales [2003] em termos de eficiência e aplicabilidade prática.

O artigo Poldi & Arenales [2003], principal referência para este estudo, aborda o uso de métodos heurísticos para encontrar soluções inteiras no Problema de Corte de Estoque Unidimensional, classificando-os em heurísticas construtivas e residuais. As heurísticas construtivas, como FFD (*First-Fit-Decreasing*) e Gulosa, constroem soluções de forma progressiva, atendendo às demandas sem gerar excessos. Já as heurísticas residuais utilizam soluções fracionárias obtidas por otimização linear relaxada como

base, ajustando-as para alcançar soluções inteiras.

Os experimentos computacionais envolveram 360 instâncias geradas aleatoriamente, baseadas no modelo de Gau & Wäscher [1995]. Os resultados mostraram que as heurísticas residuais superaram as construtivas em minimizar perdas, mesmo em cenários de baixa demanda, contrariando a ideia predominante de que heurísticas construtivas seriam mais adequadas para tais situações. A heurística "Nova"destacou-se entre as residuais, apresentando as menores perdas e o menor número médio de padrões de corte.

O estudo também enfatizou a eficiência do método de geração de colunas de Gilmore & Gomory [1961], usado para resolver o problema relaxado. As heurísticas residuais, embora mais custosas computacionalmente, demonstraram viabilidade prática superior às construtivas, principalmente em problemas com demandas limitadas.

Capítulo 4

Descrição das Heurísticas

Este capítulo descreve as abordagens desenvolvidas para resolver o problema de corte de estoque unidimensional inteiro. As soluções foram implementadas e testadas em três etapas principais. A primeira etapa, detalhada na Seção 4.1, apresenta a heurística gulosa proposta, incluindo suas justificativas e estrutura. A segunda etapa, descrita na Seção 4.2, aborda a busca local, enquanto a terceira etapa, apresentada na Seção 4.3, detalha a meta-heurística de busca tabu, que visa aprimorar o desempenho das heurísticas anteriores,e , por fim, a Seção 4.4 mostra a implementação do algoritmo genético.

Para todas as abordagens, foi utilizado um gerador aleatório descrito na Seção 2.2, que cria instâncias com base em modelos presentes em trabalhos anteriores, como Poldi & Arenales [2003] e Gau & Wäscher [1995]. A Tabela 4.1 apresenta exemplos comparativos das instâncias geradas neste estudo em relação às descritas na literatura, destacando as diferenças e semelhanças nas características dessas instâncias.

Os experimentos foram realizados na plataforma Google Colab Google [2023], utilizando o ambiente de desenvolvimento Python Python [2023]. O Google Colab oferece recursos computacionais gratuitos, permitindo a execução de algoritmos em uma instância padrão equipada com processador Intel Xeon de 2.20 GHz, 12 GB de memória RAM e sistema operacional Linux Ubuntu 18.04.

Para avaliar o desempenho das heurísticas, foram definidas 18 classes de teste, combinando diferentes valores para o número de tipos de barras (K), o número de tipos de itens (m) e os intervalos de tamanho dos itens (**P** para pequenos e **M** para médios). Cada classe foi testada com 20 instâncias distintas, totalizando 360 experimentos. As instâncias foram geradas previamente pela metodologia descrita, permitindo a aplicação das heurísticas em cenários variados e controlados.

Classes	Poldi & Arenales [2003] Boleta et al. [2005] Este Estudo			Heis et al. [2006]		
C1	3	5	P	3	5	P
C2	3	5	M	3	5	M
C3	3	20	P	3	20	P
C4	3	20	M	3	20	M
C5	3	40	P			
C6	3	40	M			
C7	5	10	P	5	10	P
C8	5	10	M	5	10	M
C9	5	20	P	5	20	P
C10	5	20	M	5	20	M
C11	5	40	P			
C12	5	40	M			
C13	7	10	P	7	10	P
C14	7	10	M	7	10	M
C15	7	20	P	7	20	P
C16	7	20	M	7	20	M
C17	7	40	P			
C18	7	40	M			

Tabela 4.1. Resumo das instâncias utilizadas nesse artigo e nos artigos relacionados. Fonte: Autória Própria.

4.1 Heurística Gulosa

Nesta seção, apresentamos a heurística gulosa desenvolvida para resolver o problema de corte de estoque unidimensional inteiro. A abordagem baseia-se na minimização do desperdício ao realizar cortes nas barras disponíveis no estoque, atendendo às demandas de itens com diferentes comprimentos. O pseudocódigo completo pode ser visualizado na Figura 4.1, e seu funcionamento está descrito detalhadamente a seguir.

A heurística gulosa segue uma lógica simples e eficaz, priorizando o corte dos maiores itens primeiro para maximizar o aproveitamento das barras. Os principais passos da heurística são:

1. **Ordenação dos itens**: Os itens são organizados em ordem decrescente de comprimento, priorizando o corte daqueles que ocupam mais espaço.

2. **Processamento de cada barra**: Cada barra disponível no estoque é utilizada até que sua capacidade restante seja insuficiente para cortar qualquer item ainda em demanda.

3. **Atualização da demanda**: A cada corte realizado, a demanda do item correspondente é reduzida, e o comprimento restante da barra é atualizado.

4. **Cálculo do desperdício**: Após processar cada barra, o comprimento não utilizado é registrado como desperdício. Esse valor é acumulado ao longo de todas as barras processadas.

Essa abordagem busca simplicidade e rapidez na obtenção de uma solução inicial, sendo um ponto de partida fundamental para aplicações subsequentes de técnicas mais sofisticadas.

```
FUNÇÃO heuristica_corte_barras(instancia):
    // Obter os dados da instância
    comprimentos_barras = instancia['comprimentos_barras']
    disponibilidade_barras = instancia['disponibilidade_barras']
    comprimentos_itens = instancia['comprimentos_itens']
    demandas = instancia['demandas']
    // Inicializar variável para o desperdício total
    desperdicio_total = 0
    // Ordenar os itens por comprimento em ordem decrescente
    itens = ordenar(comprimentos_itens e demandas por comprimento em ordem decrescente)
    // Para cada tipo de barra no estoque
    Para cada (comprimento_barra, disponibilidade) em comprimentos_barras e disponibilidade_barras:
        Para i de 1 até disponibilidade faça:  // Para cada unidade disponível dessa barra
            comprimento_restante = comprimento_barra
            // Tentar cortar os itens na barra atual, respeitando a demanda
            Para cada (comprimento_item, demanda) em itens:
                Enquanto demanda > 0 E comprimento_item <= comprimento_restante:
                    comprimento_restante -= comprimento_item  // Corte do item na barra
                    demanda -= 1  // Reduzir a demanda do item cortado
            // Adicionar o comprimento não utilizado (desperdício) da barra atual
            desperdicio_total += comprimento_restante
    // Retornar o desperdício total
    Retornar desperdicio_total
```

Figura 4.1. Pseudocódigo da heurística gulosa proposta para este estudo. Fonte: Autória Própria.

4.2 Busca Local

Nesta seção, apresentamos a abordagem de busca local desenvolvida para melhorar as soluções iniciais obtidas pela heurística gulosa. A busca local é uma técnica amplamente utilizada na otimização combinatória, especialmente em problemas complexos como o problema de corte de estoque unidimensional inteiro. Essa estratégia explora o espaço de soluções por meio de modificações incrementais em uma solução inicial, buscando melhorias no valor da função objetivo [Aarts & Lenstra [2003]].

A busca local neste estudo foi implementada considerando duas estratégias de exploração do espaço de vizinhança: a **Primeira Melhora** proposta na Subseção 4.2.2 e a **Melhor Melhora** proposta na Subseção 4.2.3. Na estratégia de primeira melhora, o algoritmo aceita imediatamente a primeira solução vizinha que melhora a solução corrente, permitindo rápidas mudanças no espaço de busca. Já na estratégia de melhor melhora, todas as soluções vizinhas são avaliadas, e o algoritmo seleciona a melhor delas para ser a nova solução corrente, resultando em uma busca mais exaustiva e detalhada.

Ambas as estratégias utilizam uma função de **Avaliação da Solução** descrita na Subseção 4.2.1, que desempenha um papel central na busca local. Essa função calcula a qualidade de uma solução com base no comprimento de barras utilizadas e no desperdício gerado. Assim, a busca local procura reduzir o desperdício e, consequentemente, melhorar a eficiência do corte.

A aplicação da busca local se baseia em duas vizinhanças principais:

1. **Troca de itens**: Modifica a alocação de itens entre barras, buscando combinações que reduzam o desperdício.

2. **Reorganização de itens**: Altera a ordem dos cortes realizados em uma mesma barra, explorando possibilidades de melhor aproveitamento.

A busca local é um passo fundamental para refinar as soluções iniciais, oferecendo ganhos substanciais de eficiência e qualidade antes de aplicar técnicas mais sofisticadas, como a busca tabu.

4.2.1 Avaliação da Solução

A função *avaliar solucao* calcula o desperdício total para uma solução (alocação de itens às barras). A função segue os seguintes passos:

1. **Inicialização do desperdício**: O desperdício total é inicializado com o valor zero.

2. **Iteração sobre as barras**: Para cada barra b utilizada no estoque, realiza-se a seguinte operação:

   a) **Soma dos itens**: Soma-se os comprimentos dos itens alocados na barra b.

   b) **Cálculo do desperdício individual**: Subtrai-se a soma dos comprimentos dos itens do comprimento total disponível da barra, obtendo o desperdício para aquela barra.

3. **Acúmulo do desperdício**: O desperdício individual de cada barra é somado ao desperdício total.

4. **Resultado final**: O desperdício total acumulado é retornado como o valor de avaliação da solução.

Essa função é essencial para verificar a qualidade de uma solução e comparar vizinhos. O pseudocódigo da função é mostrado na Figura 4.2.

```
Função avaliar_solucao(alocacao, instancia):
  // Inicializa a variável que acumula o desperdício total.
  desperdicio_total ← 0

  // Itera sobre cada barra (bin) na alocação.
  Para cada idx_barra de 0 até tamanho(alocacao) - 1 faça:
    // Inicializa o comprimento usado na barra atual.
    comprimento_usado ← 0
    // Itera sobre cada item alocado na barra atual.
    Para cada item em alocacao[idx_barra] faça:
      // Soma o comprimento do item ao comprimento total usado na barra.
      comprimento_usado ← comprimento_usado + instancia["comprimentos_itens"][item]
    Fim Para

    // Obtém o comprimento disponível da barra atual a partir da instância.
    comprimento_disponivel ← instancia["comprimentos_barras"][idx_barra]
    // Calcula o desperdício na barra atual (se positivo, senão 0).
    desperdicio ← máximo(0, comprimento_disponivel - comprimento_usado)
    // Acumula o desperdício da barra atual ao desperdício total.
    desperdicio_total ← desperdicio_total + desperdicio
  Fim Para

  // Retorna o desperdício total calculado.
  Retorne desperdicio_total
Fim Função
```

Figura 4.2. Pseudocódigo da função *avaliar solucao* utilizada nas propostas de buscas locais. Fonte: Autória Própria.

4.2.2 Primeira Melhora

A busca local com primeira melhora é uma técnica de otimização em que partimos de uma solução inicial e exploramos suas soluções vizinhas (conjunto de soluções próximas). A busca continua até que uma solução vizinha, com um desperdício menor do que a solução atual, seja encontrada. Quando isso ocorre, a nova solução é adotada, e o processo é interrompido naquele ciclo. No contexto do problema de corte de estoque, essa estratégia pode ser aplicada em duas variações: *troca de itens* entre barras e *reorganização de itens* dentro de uma mesma barra.

1. **Troca de Itens:** A estratégia de troca de itens tenta reduzir o desperdício ao mover itens entre barras diferentes, verificando se é possível alocar um item de

uma barra i para uma barra k (onde $k \neq i$) sem exceder o comprimento da barra k. As etapas para a execução dessa estratégia são:

a) **Inicialização:** A função recebe a instância do problema e a alocação inicial dos itens nas barras.

b) **Iteração sobre as barras e itens:** Para cada barra i, a função tenta mover um item j para outra barra k (onde $k \neq i$) dentro da instância, gerando uma nova solução vizinha.

c) **Verificação de espaço disponível:** Antes de mover um item, é verificado se a barra k tem espaço suficiente para o item.

d) **Avaliação do desperdício:** Após cada troca, o desperdício total da solução é avaliado utilizando a função *avaliar solucao*, proposta na Figura 4.2. Se o desperdício diminuir em relação à solução anterior, a nova solução é aceita e o processo de busca é encerrado.

e) **Resultado final:** A solução com o menor desperdício encontrado na primeira troca é retornada.

2. **Reorganização dos Itens:** A reorganização tenta melhorar a compactação dos itens dentro de uma mesma barra, alterando a ordem dos itens. As etapas são:

a) **Inicialização:** A função recebe a instância do problema e a alocação inicial dos itens nas barras.

b) **Iteração sobre as barras:** Para cada barra i que contém mais de um item, a função tenta reorganizar os itens de maneira a reduzir o desperdício.

c) **Reorganização dos itens:** A reorganização é feita movendo um item de qualquer posição para a primeira posição da lista da barra (`insert(0, item)`), buscando melhorar a compactação.

d) **Avaliação do desperdício:** Após reorganizar os itens, o desperdício total da solução é avaliado utilizando a função *avaliar solucao*, proposta na Figura 4.2. Se a reorganização resultar em menor desperdício, a nova solução é aceita e o processo de busca é encerrado.

e) **Resultado final:** A solução com o menor desperdício encontrado na primeira reorganização é retornada.

A estratégia primeira melhora evita explorar toda a vizinhança, o que contribui para uma busca mais eficiente em termos de tempo de execução. Assim, a busca local é interrompida assim que uma melhoria é identificada, acelerando o processo de

otimização. As Figuras 4.3 e 4.4 apresentam os pseudocódigos das heurísticas de busca local para as vizinhanças de troca e reorganização de itens, respectivamente.

```
Função primeira_melhora_troca_itens(instancia, alocacao_inicial):
  // Inicializa a melhor solução encontrada com a solução inicial.
  melhor_solucao ← alocacao_inicial
  // Calcula o desperdício da solução inicial.
  melhor_desperdicio ← avaliar_solucao(alocacao_inicial, instancia)

  // Itera sobre todas as barras.
  Para cada i de 0 até tamanho(alocacao_inicial) - 1 faça:
    // Itera sobre todos os itens de cada barra.
    Para cada j de 0 até tamanho(alocacao_inicial[i]) - 1 faça:
      // Itera sobre todas as outras barras.
      Para cada k de 0 até tamanho(alocacao_inicial) - 1 faça:
        // Verifica se i e k são diferentes (para evitar mover um item para a mesma barra).
        Se i ≠ k então:
          // Cria uma cópia da alocação para explorar um vizinho.
          novo_vizinho ← cópia profunda de alocacao_inicial
          // Remove o item j da barra i.
          item ← remover item na posição j de novo_vizinho[i]
          // Verifica se há espaço suficiente na barra k para o item.
          Se instancia["comprimentos_barras"][k] ≥ instancia["comprimentos_itens"][item] então:
            // Adiciona o item à barra k.
            adicionar item a novo_vizinho[k]
            // Calcula o desperdício do novo vizinho.
            desperdicio_vizinho ← avaliar_solucao(novo_vizinho, instancia)
            // Se o desperdício do vizinho é menor que o melhor desperdício encontrado até agora:
            Se desperdicio_vizinho < melhor_desperdicio então:
              // Retorna o novo vizinho (primeira melhora).
              Retorne novo_vizinho, desperdicio_vizinho
            Fim Se
          Fim Se
        Fim Se
      Fim Para
    Fim Para
  Fim Para

  // Se nenhuma solução melhor foi encontrada, retorna a solução inicial.
  Retorne melhor_solucao, melhor_desperdicio
Fim Função
```

Figura 4.3. Pseudocódigo da heurística de busca local com primeira melhora utilizando a vizinhança de troca de itens. Fonte: Autoria Própria.

```
Função primeira_melhora_reorganizacao(instancia, alocacao_inicial):
  // Inicializa a melhor solução com a solução inicial.
  melhor_solucao ← alocacao_inicial
  // Calcula o desperdício da solução inicial.
  melhor_desperdicio ← avaliar_solucao(alocacao_inicial, instancia)

  // Itera sobre cada barra.
  Para cada i de 0 até tamanho(alocacao_inicial) - 1 faça:
    // Verifica se a barra contém mais de um item (necessário para reorganização).
    Se tamanho(alocacao_inicial[i]) > 1 então:
      // Itera sobre cada item na barra.
      Para cada j de 0 até tamanho(alocacao_inicial[i]) - 1 faça:
        // Cria uma cópia da solução para explorar a vizinhança.
        novo_vizinho ← cópia profunda de alocacao_inicial
        // Remove o item j da barra i.
        item ← remover item na posição j de novo_vizinho[i]
        // Insere o item no início da barra i (reorganização).
        inserir item na posição 0 de novo_vizinho[i]
        // Calcula o desperdício do novo vizinho.
        desperdicio_vizinho ← avaliar_solucao(novo_vizinho, instancia)
        // Se o desperdício do vizinho é menor que o melhor desperdício encontrado até agora:
        Se desperdicio_vizinho < melhor_desperdicio então:
          // Retorna o novo vizinho (primeira melhora).
          Retorne novo_vizinho, desperdicio_vizinho
        Fim Se
      Fim Para
    Fim Se
  Fim Para

  // Se nenhuma solução melhor foi encontrada, retorna a solução inicial.
  Retorne melhor_solucao, melhor_desperdicio
Fim Função
```

Figura 4.4. Pseudocódigo da heurística de busca local com primeira melhora utilizando a vizinhança de reorganização de itens. Fonte: Autoria Própria.

4.2.3 Melhor Melhora

A heurística de melhor melhora é uma técnica de busca local que explora as vizinhanças de uma solução para encontrar uma melhor alocação de itens nas barras. Diferentemente da heurística de primeira melhora, que aceita a primeira solução vizinha que apresenta uma melhora, a heurística de melhor melhora busca explorar todas as possibilidades dentro da vizinhança e seleciona a melhor solução possível. Para o problema de corte de estoque unidimensional, duas variações de vizinhanças são utilizadas: *troca de itens* entre barras e *reorganização dos itens* dentro das barras.

1. **Troca de Itens:** A estratégia de troca de itens tenta reduzir o desperdício ao mover itens entre barras diferentes, verificando se é possível alocar um item de uma barra i para uma barra k (onde $k \neq i$) sem exceder o comprimento da barra k. As etapas para a execução dessa estratégia são:

   a) **Inicialização:** A função recebe a instância do problema e a alocação inicial dos itens nas barras.

   b) **Iteração sobre as barras e itens:** Para cada barra i, a função tenta mover um item j para outra barra k (onde $k \neq i$) dentro da instância, gerando uma nova solução vizinha.

   c) **Verificação de espaço disponível:** Antes de mover um item, é verificado se a barra k tem espaço suficiente para o item.

   d) **Avaliação do desperdício:** Após cada troca, o desperdício total da solução é avaliado utilizando a função *avaliar solucao*, proposta na Figura 4.2. Se o desperdício diminuir em relação à solução anterior, a nova solução é mantida.

   e) **Resultado final:** O processo continua até que todas as trocas possíveis tenham sido tentadas. A solução com o menor desperdício encontrado é retornada.

2. **Reorganização dos Itens:** A reorganização tenta melhorar a compactação dos itens dentro de uma mesma barra, alterando a ordem dos itens. As etapas são:

   a) **Inicialização:** A função recebe a instância do problema e a alocação inicial dos itens nas barras.

   b) **Iteração sobre as barras:** Para cada barra i que contém mais de um item, a função tenta reorganizar os itens de maneira a reduzir o desperdício.

   c) **Reorganização dos itens:** A reorganização é feita movendo um item de qualquer posição para a primeira posição da lista da barra (`insert(0, item)`), buscando melhorar a compactação.

d) **Avaliação do desperdício:** Após reorganizar os itens, o desperdício total da solução é avaliado utilizando a função *avaliar solucao*, proposta na Figura 4.2. Se a reorganização resultar em menor desperdício, a nova solução é mantida.

e) **Resultado final:** O processo continua até que todas as barras com mais de um item sejam reorganizadas, e a solução com o menor desperdício é retornada.

A heurística de melhor melhora realiza uma exploração mais abrangente das soluções vizinhas, buscando sempre a solução que minimiza o desperdício total. Como resultado, essa abordagem tende a fornecer soluções mais refinadas do que a heurística de primeira melhora, embora em maior tempo computacional. As Figuras 4.5 e 4.6 apresentam os pseudocódigos das heurísticas de busca local para as vizinhanças de troca e reorganização de itens, respectivamente.

```
Função melhor_melhora_troca_itens(instancia, alocacao_inicial):
  // Inicializa a melhor solução com a solução inicial.
  melhor_solucao ← alocacao_inicial
  // Calcula o desperdício da solução inicial.
  melhor_desperdicio ← avaliar_solucao(alocacao_inicial, instancia)

  // Itera sobre todas as barras.
  Para cada i de 0 até tamanho(alocacao_inicial) - 1 faça:
    // Itera sobre todos os itens em cada barra.
    Para cada j de 0 até tamanho(alocacao_inicial[i]) - 1 faça:
      // Itera sobre todas as outras barras.
      Para cada k de 0 até tamanho(alocacao_inicial) - 1 faça:
        // Evita trocar o item para a mesma barra.
        Se i ≠ k então:
          // Cria uma cópia da alocação para explorar um vizinho.
          novo_vizinho ← cópia profunda de alocacao_inicial
          // Remove o item j da barra i.
          item ← remover item na posição j de novo_vizinho[i]
          // Verifica se há espaço suficiente na barra k para o item.
          Se instancia["comprimentos_barras"][k] ≥ instancia["comprimentos_itens"][item] então:
            // Adiciona o item à barra k.
            adicionar item a novo_vizinho[k]
            // Calcula o desperdício do novo vizinho.
            desperdicio_vizinho ← avaliar_solucao(novo_vizinho, instancia)
            // Se o desperdício do vizinho é menor que o melhor desperdício encontrado até agora:
            Se desperdicio_vizinho < melhor_desperdicio então:
              // Atualiza a melhor solução e o melhor desperdício.
              melhor_solucao ← novo_vizinho
              melhor_desperdicio ← desperdicio_vizinho
            Fim Se
          Fim Se
        Fim Se
      Fim Para
    Fim Para
  Fim Para

  // Retorna a melhor solução e o melhor desperdício encontrados.
  Retorne melhor_solucao, melhor_desperdicio
Fim Função
```

Figura 4.5. Pseudocódigo da heurística de busca local para melhor melhora utilizando a vizinhança de troca de itens. Fonte: Autoria Própria.

```
Função melhor_melhora_reorganizacao(instancia, alocacao_inicial):
  // Inicializa a melhor solução com a solução inicial.
  melhor_solucao ← alocacao_inicial
  // Calcula o desperdício inicial.
  melhor_desperdicio ← avaliar_solucao(alocacao_inicial, instancia)

  // Itera sobre cada barra.
  Para cada i de 0 até tamanho(alocacao_inicial) - 1 faça:
    // Verifica se a barra contém mais de um item para reorganizar.
    Se tamanho(alocacao_inicial[i]) > 1 então:
      // Itera sobre cada item na barra.
      Para cada j de 0 até tamanho(alocacao_inicial[i]) - 1 faça:
        // Cria uma cópia da solução para explorar um vizinho.
        novo_vizinho ← cópia profunda de alocacao_inicial
        // Remove o item j da barra i.
        item ← remover item na posição j de novo_vizinho[i]
        // Insere o item no início da barra i.
        inserir item na posição 0 de novo_vizinho[i]
        // Calcula o desperdício do novo vizinho.
        desperdicio_vizinho ← avaliar_solucao(novo_vizinho, instancia)
        // Se o desperdício do vizinho é menor, atualiza a melhor solução.
        Se desperdicio_vizinho < melhor_desperdicio então:
          melhor_solucao ← novo_vizinho
          melhor_desperdicio ← desperdicio_vizinho
        Fim Se
      Fim Para
    Fim Se
  Fim Para

  // Retorna a melhor solução e o seu desperdício.
  Retorne melhor_solucao, melhor_desperdicio
Fim Função
```

Figura 4.6. Pseudocódigo da heurística de busca local para melhor melhora utilizando a vizinhança de reorganização de itens. Fonte: Autoria Própria.

4.3 Busca Tabu

A busca tabu é uma técnica heurística de otimização combinatória amplamente utilizada em problemas de grande escala, onde o espaço de soluções é vasto e não pode ser explorado exaustivamente. Proposta por Glover [1986], essa técnica baseia-se na ideia de mover-se em torno de uma solução inicial por meio de pequenas modificações, utilizando uma estrutura de memória chamada *lista tabu*. O objetivo principal da busca tabu é escapar de ótimos locais, permitindo que o algoritmo explore mais eficientemente o espaço de soluções.

Esse método é particularmente eficaz em problemas de otimização combinatória, como o problema de corte de estoque, onde as soluções possíveis são numerosas e uma abordagem exata seria computacionalmente inviável. No contexto do problema de corte de estoque unidimensional inteiro, o algoritmo busca alocar os itens de forma eficiente nas barras disponíveis, minimizando o desperdício de material e atendendo às demandas de maneira adequada.

A busca tabu utiliza o conceito de vizinhança, onde uma solução vizinha é gerada aplicando-se uma pequena modificação na solução atual. Além disso, o algoritmo mantém uma memória das soluções recentemente exploradas para impedir a volta a soluções já visitadas, utilizando a lista tabu. Esse mecanismo permite que o algoritmo explore o espaço de soluções de maneira diversificada, evitando ciclos e aumentando as chances de encontrar uma solução melhor.

Para o problema de corte de estoque, foram implementadas duas vizinhanças principais, descritas a seguir:

- **Troca de Itens:** Nessa vizinhança, dois itens são trocados entre diferentes barras de modo a minimizar o desperdício.

- **Reorganização de Itens:** A reorganização visa alterar a ordem dos itens dentro das barras para uma alocação mais eficiente.

O algoritmo de busca tabu funciona da seguinte maneira:

1. **Geração da solução inicial:** O algoritmo começa com uma alocação inicial dos itens nas barras, gerando uma solução inicial.

2. **Geração de vizinhos:** A partir da solução atual, são gerados vizinhos por meio de pequenas modificações, como a troca de itens entre barras ou a reorganização dos itens dentro das barras.

3. **Avaliação de vizinhos:** Cada vizinho gerado é avaliado com base no desperdício total de material que é avaliado utilizando a função *avaliar solucao*, proposta na Figura 4.2, buscando minimizar o desperdício e maximizar a utilização das barras.

4. **Seleção do melhor vizinho:** O vizinho com o menor desperdício é selecionado como a nova solução. Se essa solução ainda não foi visitada (não está na lista tabu), ela é adicionada à lista.

5. **Atualização da lista tabu:** A solução mais antiga é removida da lista tabu e a nova solução é adicionada. O tamanho da lista tabu é controlado para equilibrar a exploração de soluções novas e a não revisitação de soluções antigas.

6. **Repetição do processo:** O processo é repetido por um número fixo de iterações ou até que um critério de parada seja atendido (por exemplo, um número máximo de iterações ou uma melhoria mínima na solução).

A busca tabu será aplicada de forma iterativa, gerando vizinhos e mantendo o controle das soluções visitadas, garantindo que o algoritmo explore uma gama ampla de possíveis soluções sem cair em ciclos. O pseudocódigo do algoritmo pode ser visualizado na Figura 4.7.

A seguir, descrevem-se as duas vizinhanças implementadas na busca tabu para o problema de corte de estoque.

```
Função BuscaTabu(instancia, soluçãoInicial, funçãoVizinhos, iteraçõesMax, tamanhoTabu):
 // Algoritmo de Busca Tabu para otimização.

 melhorSolução ← soluçãoInicial // Inicializa a melhor solução encontrada até o momento.
 melhorCusto ← avaliarCusto(soluçãoInicial, instancia) // Calcula o custo da solução inicial.
 soluçãoAtual ← soluçãoInicial // Inicializa a solução atual.
 listaTabu ← uma fila de tamanho máximo tamanhoTabu  // Lista Tabu que armazena soluções visitadas recentemente (LI

 Para i de 1 até iteraçõesMax: // Loop principal de iterações.
   vizinhos ← funçãoVizinhos(instancia, soluçãoAtual) // Gera os vizinhos da solução atual.
   melhorVizinho ← nulo // Inicializa o melhor vizinho encontrado na iteração atual.
   melhorCustoVizinho ← infinito // Inicializa o custo do melhor vizinho.

   Para cada vizinho em vizinhos: // Itera sobre os vizinhos gerados.
     Se vizinho não está em listaTabu:  // Verifica se o vizinho está na lista tabu. Critério Tabu básico.
       custoVizinho ← avaliarCusto(vizinho, instancia) // Calcula o custo do vizinho.
       Se custoVizinho < melhorCustoVizinho: // Se o custo do vizinho é melhor que o melhor custo encontrado até ag
         melhorVizinho ← vizinho // Atualiza o melhor vizinho.
         melhorCustoVizinho ← custoVizinho // Atualiza o melhor custo.

   Se melhorVizinho ≠ nulo: // Se um vizinho melhor foi encontrado.
     soluçãoAtual ← melhorVizinho // Atualiza a solução atual.
     adicionar melhorVizinho à listaTabu // Adiciona o melhor vizinho à lista tabu.

     Se melhorCustoVizinho < melhorCusto: // Se o custo do melhor vizinho é melhor que o melhor custo global.
       melhorSolução ← melhorVizinho // Atualiza a melhor solução global.
       melhorCusto ← melhorCustoVizinho // Atualiza o melhor custo global.

 Retornar melhorSolução, melhorCusto // Retorna a melhor solução encontrada e seu custo.
```

Figura 4.7. Pseudocódigo da heurística de busca tabu. Fonte: Autoria Própria.

1. **Troca de Itens**

   A vizinhança de troca de itens é implementada na função *vizinhanca troca itens*, cujo pseudocódigo pode ser visualizado na Figura 4.8. Nessa vizinhança, o algoritmo gera vizinhos da solução atual trocando itens entre diferentes barras. A modificação visa melhorar a utilização do espaço disponível e, consequentemente, reduzir o desperdício de material. O código segue os seguintes passos:

   a) **Iteração sobre as barras e itens:** Para cada barra e item dentro dela, são feitas tentativas de troca de itens entre barras diferentes.

   b) **Criação de novos vizinhos:** Cada vez que uma troca é realizada, uma cópia profunda da alocação atual é criada e a troca é aplicada.

   c) **Verificação de viabilidade:** Antes de realizar a troca, verifica-se se a

barra de destino tem espaço suficiente para acomodar o item trocado.

d) **Armazenamento de vizinhos:** Os novos vizinhos são armazenados e retornados para serem avaliados.

O objetivo dessa vizinhança é explorar diferentes combinações de itens alocados nas barras para reduzir o desperdício.

```
Função vizinhancaTrocaItens(instancia, alocaçãoInicial):
  // Gera vizinhos trocando itens entre diferentes barras.

  vizinhos ← lista vazia

  Para cada barra_i de 0 até tamanho(alocaçãoInicial) - 1:
    Para cada item_j de 0 até tamanho(alocaçãoInicial[barra_i]) - 1:
      Para cada barra_k de 0 até tamanho(alocaçãoInicial) - 1:
        Se barra_i ≠ barra_k: // Evita trocar itens na mesma barra
          novoVizinho ← cópia profunda de alocaçãoInicial
          item ← remover item_j da barra barra_i em novoVizinho // Remove o item da barra de origem
          Se espaçoSuficiente(instancia, item, barra_k): // Verifica se há espaço suficiente na barra destino
            adicionar item à barra barra_k em novoVizinho  // Adiciona o item à barra de destino
            adicionar novoVizinho à vizinhos // Adiciona o novo vizinho à lista

  Retornar vizinhos
```

Figura 4.8. Pseudocódigo da vizinhança de troca de itens para a busca tabu. Fonte: Autoria Própria.

2. **Reorganização de Itens**

A vizinhança de reorganização é implementada na função *vizinhanca reorganizacao*, cujo pseudocódigo pode ser visualizado na Figura 4.9. Nesse tipo de vizinhança, o algoritmo tenta melhorar a solução alterando a ordem dos itens dentro das barras. O objetivo é explorar diferentes disposições dos itens dentro de cada barra, o que pode levar a uma melhor utilização do espaço e, portanto, à redução do desperdício de material. O código segue os seguintes passos:

a) **Iteração sobre as barras:** Para cada barra que contém mais de um item, o algoritmo tenta reorganizar os itens dentro dessa barra.

b) **Reorganização dos itens:** Os itens são removidos e reinseridos em diferentes posições dentro da barra, alterando a ordem dos itens.

c) **Armazenamento de vizinhos:** Cada nova configuração gerada pela reorganização é armazenada e retornada para ser avaliada.

O objetivo dessa vizinhança é melhorar a alocação dos itens dentro das barras, buscando reduzir o desperdício de material.

```
Função vizinhancaReorganização(instancia, alocaçãoInicial):
  // Gera vizinhos reorganizando os itens dentro de cada barra.

  vizinhos ← lista vazia

  Para cada barra_i de 0 até tamanho(alocaçãoInicial) - 1:
    Se tamanho(alocaçãoInicial[barra_i]) > 1: // Somente barras com mais de um item
      Para cada item_j de 0 até tamanho(alocaçãoInicial[barra_i]) - 1:
        novoVizinho ← cópia profunda de alocaçãoInicial
        item ← remover item_j da barra barra_i em novoVizinho // Remove o item da barra
        inserir item em uma posição aleatória na barra barra_i em novoVizinho // Insere o item em uma posição aleató
        adicionar novoVizinho à vizinhos // Adiciona o novo vizinho à lista

  Retornar vizinhos
```

Figura 4.9. Pseudocódigo da vizinhança de reorganização de itens para a busca tabu. Fonte: Autoria Própria.

A busca tabu será aplicada iterativamente, utilizando tanto a troca de itens quanto a reorganização dos itens. O algoritmo gerará vizinhos e manterá o controle das soluções visitadas, garantindo que o espaço de soluções seja explorado de forma eficaz sem cair em ciclos.

4.4 Algortimo Genético

Esta seção apresenta um algoritmo genético (AG) para resolver o problema unidimensional de de corte de barras. O algoritmo utiliza uma representação onde cada indivíduo na população é uma permutação dos índices dos itens, representando a ordem em que os itens são colocados nos bins. A aptidão de um indivíduo é determinada pelo espaço desperdiçado total após a colocação dos itens de acordo com sua permutação, utilizando a função de decodificar indivíduo mostrada na Figura 4.10. Essa função simula o processo de empacotamento, atribuindo itens aos bins sequencialmente com base em seu comprimento e capacidade disponível.

O AG utiliza um operador de Crossover Ordenado (OX) (Figura 4.11) para gerar descendentes. O OX preserva a ordem relativa de uma subsequência de um dos pais, promovendo a exploração de estruturas de solução promissoras, mantendo ao mesmo tempo alguma diversidade. Um operador de mutação simples de troca (Figura 4.12) é implementado para introduzir variações locais no espaço de solução, evitando a convergência prematura. A seleção por torneio (Figura 4.13) é empregada para selecionar

```
 INPUT: permutacao (ordem dos itens), instancia (comprimentos dos itens, capacidades dos bins)
OUTPUT: alocacao (itens em cada bin), desperdicio

K = numero de bins
capacidades = copia das capacidades dos bins  // Mantem as capacidades originais
alocacao = lista de K listas vazias (bins)

PARA CADA item em permutacao:
  comprimento_item = comprimento do item atual
  PARA CADA bin i de 0 ate K-1:
    SE comprimento_item <= capacidades[i]:
      adicionar item a alocacao[i]
      capacidades[i] = capacidades[i] - comprimento_item
      SAIR (ir para o próximo item) // Item alocado, passa para o próximo

desperdicio = soma das capacidades restantes
RETORNAR alocacao, desperdicio
```

Figura 4.10. Pseudocódigo de decodificar individuo. Fonte: Autoria Própria.

os pais, adicionando estocasticidade ao processo de busca. O elitismo é incorporado para garantir que a melhor solução encontrada até o momento seja preservada entre as gerações.

A função algoritmo genético mostrada na Figura 4.14 orquestra o AG, aplicando iterativamente seleção, crossover e mutação para evoluir a população. Os parâmetros do algoritmo, incluindo o tamanho da população, o número de gerações, a taxa de crossover, a taxa de mutação e o tamanho do torneio, influenciam seu desempenho e podem ser ajustados para resultados ótimos com base na instância específica do problema. Pesquisas futuras se concentrarão na exploração de operadores alternativos de crossover e mutação, controle adaptativo de parâmetros e na incorporação de heurísticas de busca local para potencialmente melhorar a qualidade e a eficiência da solução. A eficácia dessa abordagem será avaliada por meio de experimentos computacionais em instâncias de referência do problema unidimensional de corte de barras.

```
 INPUT: pai1, pai2 (permutações)
OUTPUT: filho (permutação)

tamanho = comprimento dos pais
a, b = dois índices aleatórios, a < b
filho = lista de tamanho None values

filho[a:b+1] = pai1[a:b+1] // Copia uma subsequência do pai 1
pos = (b + 1) % tamanho // índice circular para percorrer o vetor

PARA CADA gene em pai2:
  SE gene NAO esta em filho:
    filho[pos] = gene
    pos = (pos + 1) % tamanho

RETORNAR filho
```

Figura 4.11. Pseudocódigo de crossover ordenado. Fonte: Autoria Própria.

```
 INPUT: individuo (permutação)
OUTPUT: individuo_mutado

a, b = dois índices aleatórios distintos
TROCAR individuo[a] e individuo[b]
RETORNAR individuo
```

Figura 4.12. Pseudocódigo de mutação troca. Fonte: Autoria Própria.

```
INPUT: populacao (lista de permutações), aptidao (lista de espaços desperdiçados), tamanho_torneio
OUTPUT: individuo_selecionado

indices = amostra aleatória de tamanho_torneio de índices da população
melhor_indice = índice com o menor valor de aptidao em indices
RETORNAR copia de populacao[melhor_indice]
```

Figura 4.13. Pseudocódigo de seleção torneio. Fonte: Autoria Própria.

```
INPUT: instancia (dados do problema), parâmetros do algoritmo
OUTPUT: melhor_individuo, melhor_desperdicio

populacao = inicializar com permutações aleatórias dos índices dos itens
melhor_individuo = None
melhor_desperdicio = infinito

PARA g = 0 ate geracoes - 1:
  aptidao = avaliar cada individuo na populacao usando decodificar_individuo
  atualizar melhor_individuo e melhor_desperdicio se necessário
  nova_populacao = [melhor_individuo] // Elitismo
  ENQUANTO tamanho de nova_populacao < tamanho_populacao:
    pai1 = selecao_torneio(populacao, aptidao, tamanho_torneio)
    pai2 = selecao_torneio(populacao, aptidao, tamanho_torneio)
    SE aleatorio < taxa_crossover:
      filho = crossover_ordenado(pai1, pai2)
    SENAO:
      filho = pai1
    SE aleatorio < taxa_mutacao:
      mutacao_troca(filho)
    adicionar filho a nova_populacao
  populacao = nova_populacao

RETORNAR melhor_individuo, melhor_desperdicio
```

Figura 4.14. Pseudocódigo do algoritmo genético, função principal. Fonte: Autoria Própria.

Capítulo 5

Resultados

5.1 Resultados entre as Heurísticas Propostas

Este capítulo apresenta os resultados obtidos a partir da implementação das heurísticas desenvolvidas no capítulo anterior para o problema de corte de estoque unidimensional inteiro. Os métodos comparados incluem a heurística gulosa, a busca local (com quatro variações: primeira melhora e melhor melhora para as vizinhanças de troca e reorganização) e a busca tabu (também utilizando vizinhanças de troca e reorganização) e o algoritmo genético. Os resultados foram analisados considerando a média de desperdício de material e o tempo de execução.

É importante destacar que algumas classes apresentaram desperdício nulo, com valor igual a zero. Esse comportamento ocorre devido à simplicidade das configurações dessas instâncias, que favorecem um desempenho ótimo para todos os métodos. Por esse motivo, tais classes têm menor relevância na comparação entre as heurísticas. Assim, essas instâncias foram consideradas, mas com peso reduzido na análise de desempenho geral dos métodos.

A Tabela 5.1 apresenta os resultados comparativos das heurísticas, com os valores da média de desperdício para cada método e classe de instâncias (C1 a C18). Os resultados do Algoritmo Genético é apresentado de forma individual na Tabela 5.3. No final das tabelas, também é exibida a quantidade de melhores resultados obtidos por cada método, a média geral e o desvio padrão, o que permite identificar o mais eficiente. Sendo assim, selecionaremos os principais resultados para a comparação dos resultados com os estudos relacionados deste trabalho.

	Método Guloso		Busca Local				Busca Tabu	
Classes	**Gulosa 1ª Ex.**	**Gulosa 2ª Ex.**	**Local P.M. T.**	**Local P.M. R.**	**Local M.M. T.**	**Local M.M. R.**	**Tabu T.**	**Tabu R.**
C1	**655,6**	914,25	**133,4**	135,85	153,95	133,65	146,15	**125,3**
C2	1088,8	**815,65**	**70,9**	96,9	84,15	97,7	**52,75**	108,65
C3	0	**93**	105,55	127,9	**99,7**	114,8	**52,2**	111,7
C4	718,85	**399,55**	53,65	97,6	**50,15**	103,75	0	**101,5**
C5	**0**	**0**	106,25	100,4	99,45	**96,2**	0	**104,4**
C6	**82,65**	781,6	**38**	99	52,45	94,35	0	**85,8**
C7	214,9	**151**	238,5	**223,15**	241,75	237,2	**222,5**	223,7
C8	**1638,45**	1990	**168,35**	216,5	174,85	203,9	**55,35**	200,15
C9	**29,05**	162,8	**192,2**	228,95	215,7	205,1	**172,8**	218,35
C10	**797,3**	1162,75	**153,3**	212,2	165,55	174,85	0	**186,4**
C11	**132,5**	0	227,8	235,05	215,3	**210,05**	**62,65**	234,1
C12	**855,25**	917,05	**162,92**	215,4	167,2	212,5	0	**187,15**
C13	362,05	**249,9**	358,25	342,35	**321,75**	331,75	368,9	**323,45**
C14	**2114,8**	2130,7	275,7	294,4	**259,85**	306,3	**157,7**	313,55
C15	**150,95**	286,5	332,75	327,15	337,3	**325,9**	**286,8**	315,05
C16	1762,3	**748,3**	261,15	332	**258,7**	331,65	**9,15**	281,1
C17	**0**	**0**	343,95	**330,4**	334,95	336,4	**166,45**	327,45
C18	**901,2**	918,3	259,3	299,8	**258,4**	283,85	0	**290,5**
Qnt. Melhores Result.	**12**	8	**7**	2	6	3	**10**	8
Média Geral	**639,15**	651,19	**193,44**	217,50	193,95	211,11	**97,41**	207,68
Desvio Padrão	643,17	**620,82**	96,88	**87,57**	90,28	88,47	108,89	**84,11**

Tabela 5.1. Tabela de resultados da média de desperdício do problema de corte para cada classe de barra por cada método proposto neste estudo. Fonte: Autoria Própria. Ex.: Execução. P.M.: Primeira Melhora. M.M.:Melhor Melhora. T.: Troca de Itens. R.: Reorganização de Itens.

Além disso, o tempo de execução é um fator essencial para avaliar a aplicabilidade prática das heurísticas, especialmente em cenários onde o tempo de resposta é crítico. A Tabela 5.2 apresenta os tempos médios de execução agregados para cada método, fornecendo uma perspectiva adicional sobre a eficiência computacional.

	Método Guloso	**Busca Local P.M.**	**Busca Local M.M.**	**Busca Tabu**
Tempo Exec.(s)	**3,35**	**0,39**	**1,65**	**148,54**

Tabela 5.2. Tempo de execução dos métodos propostos em segundos. Fonte: Autoria Própria. P.M.: Primeira Melhora. M.M.:Melhor Melhora.

Algoritmo Genético		
Classes	**1ª Exec**	**2ª Exec**
C1	**145,05**	148,7
C2	**63,3**	63,8
C3	**57,75**	59,85
C4	0,15	**0,05**
C5	0	0
C6	0	0
C7	229,75	**223,75**
C8	**55,9**	60,8
C9	168,8	**156**
C10	0,5	**0,45**
C11	57,15	**57,05**
C12	0	0
C13	332,1	**326,3**
C14	173,45	**159,75**
C15	**269,1**	274,5
C16	**7**	7,25
C17	185,1	**175,05**
C18	**0,05**	0,25
Qnt. Melhores Resultados	6	**8**
Média Geral	96,95	**95,20**
Desvio Padrão	103,84	**101,67**
Tempo Exec.	75,5	**74,5**

Tabela 5.3. Tabela de resultados da média de desperdício do problema de corte para cada classe de barra pelo método de Algoritmo Genético. Fonte: Autoria Própria.

Os resultados das tabelas revelam diferenças importantes entre os métodos guloso, busca local e busca tabu:

- **Tempo de Execução**: O método guloso e as buscas locais apresentaram tempos de execução bastante reduzidos, variando de 0 a 3 segundos aproximadamente, o que reflete sua simplicidade computacional. Em contrapartida, a busca tabu teve tempos de execução significativamente maiores, excedendo 2 minutos, devido à complexidade inerente ao seu mecanismo de exploração intensiva e controle da lista tabu.

- **Desempenho dos Métodos**:

  - *Heurística Gulosa*: O método guloso apresentou o pior desempenho em termos de desperdício médio entre as instâncias testadas, com uma média geral de 639,15. Apesar disso, foi selecionada a primeira execução desse método (Gulosa 1ª Ex.) para a comparação com os estudos relacionados, dado seu papel como baseline.

  - *Busca Local*: Entre as variantes de busca local, a estratégia de primeira melhora com vizinhança de troca (Local P.M. T.) foi a mais eficiente, alcançando os melhores resultados em 7 instâncias e uma média geral de 193,44. Por isso, foi selecionada como representante das buscas locais para a comparação final. Outras variantes, como a melhor melhora com vizinhança de troca (Local M.M. T.), também tiveram um bom desempenho, obtendo melhores resultados em 6 instâncias e uma média de 193,95. Essas heurísticas mostraram grande potencial para refinar soluções em instâncias mais complexas, mesmo não superando consistentemente a busca tabu.

  - *Busca Tabu*: O método tabu destacou-se como a heurística mais competitiva, especialmente na variante com vizinhança de troca (Tabu T.), que obteve os melhores resultados em 10 instâncias. Em comparação, a variante com reorganização (Tabu R.) teve desempenho inferior, confirmando a superioridade da abordagem baseada em troca para a busca tabu. Assim, a variante Tabu T. foi selecionada como representante do método tabu para a comparação final com os estudos relacionados.

  - *Algoritmo Genético*: Os resultados do algoritmo genético evidenciam um desempenho notável, com médias gerais de desperdício de aproximadamente 96 e 95 nas duas execuções, além de uma dispersão (desvio padrão) relativamente baixa (em torno de 102). Apesar de seu tempo de execução ser maior que o das heurísticas simples e das buscas locais, ele se mostra competitivo frente à busca tabu. Essa combinação de alta qualidade na solução e tempo de processamento intermediário posiciona o algoritmo genético como uma alternativa robusta para o problema de corte de estoque.

Essa análise integrada dos resultados permite uma comparação abrangente entre os métodos, destacando a eficiência do algoritmo genético em termos de desperdício de material, mesmo considerando um tempo de execução um pouco maior. Dessa forma, a seleção do método mais adequado dependerá do *trade-off* entre qualidade da solução e exigências de tempo de processamento para cada aplicação específica.

5.2 Resultados Finais de Comparação

Os resultados finais estão apresentados na Tabela 5.4, que mostra a média do desperdício total para as diferentes classes analisadas mais o algoritmo genético proposto de forma individual na Tabela 5.3, comparando-as com os valores obtidos nos trabalhos: Estudo 1 Poldi & Arenales [2003], Estudo 2 Boleta et al. [2005] e Estudo 3 Heis et al. [2006]. Foram utilizadas as versões selecionadas das heurísticas propostas (heurística gulosa, busca local, busca tabu e algoritmo genético) para essa comparação.

Média da Perda Total						
Classes	**Estudo 1**	**Estudo 2**	**Estudo 3***	**Gulosa**	**Local P.M. T.**	**Tabu T.**
C1	177	**73,30**	105,85	655,6	133,4	146,15
C2	524	333,35	204,3	1088,8	70,9	**52,75**
C3	187	72,05	77,35	0	105,55	**52,2**
C4	842	499,85	178,7	718,85	**53,65**	0
C5	115	**87,75**		0	106,25	0
C6	852	1169,95		82,65	**38**	0
C7	125	**38,20**	68	214,9	238,5	222,5
C8	1034	372,75	127,15	1638,45	168,35	**55,35**
C9	125	52,10	46,6	**29,05**	192,2	172,8
C10	846	493,45	**144,75**	797,3	153,3	0
C11	127	66,30		132,5	227,8	**62,65**
C12	780	1494,65		855,25	**162,92**	0
C13	81	**54,00**	104,85	362,05	358,25	368,9
C14	466	143,25	**71,65**	2114,8	275,7	157,7
C15	97	51,25	**16,5**	150,95	332,75	286,8
C16	274	272,70	69,4	1762,3	261,15	**9,15**
C17	91	**51,15**		0	343,95	166,45
C18	1159	1850,15		901,2	**259,3**	0
Qnt. Melhores Resultados	0	**5**	3	1	4	**5**
Média Geral	439	398,68	101,09	639,15	193,44	**97,41**
Desvio Padrão	366,7	529,05	**52,57**	643,17	96,88	108,89

Tabela 5.4. Média das perdas totais de barras por classe. **Estudo 1**: Resultados obtidos por Poldi & Arenales [2003]. **Estudo 2**: Resultados obtidos por Boleta et al. [2005]. **Estudo 3**: Resultados obtidos por Heis et al. [2006]. Fonte: Autória Própria. P.M.: Primeira Melhora. T.: Troca de Itens. *Avaliar com cuidado o Estudo 3, pois o estudo é feito apenas com 12 classes.

A análise dos resultados deve considerar também a natureza dos métodos utilizados em cada estudo:

- O **Estudo 1** empregou um método guloso, caracterizado por decisões locais otimizadas iterativamente, mas que podem levar a soluções subótimas em problemas complexos.
- Os **Estudos 2 e 3** utilizaram algoritmos genéticos, que exploram o espaço de soluções por meio de recombinação e mutação, sendo mais adequados para instâncias maiores ou mais complexas.

A análise comparativa evidencia que, entre os métodos tradicionais (Estudos 1, 2 e 3) e as heurísticas propostas, o método guloso obteve desempenho inferior em todas

as métricas avaliadas, enquanto o Estudo 3 se destacou pela consistência, registrando a menor média geral de desperdício (101,09) e o menor desvio padrão (52,57).

Dentre as heurísticas desenvolvidas, a busca local (Local P.M. T.) apresentou desempenho intermediário, com média geral de 193,44 e baixa variabilidade (desvio padrão de 96,88). Em contrapartida, as abordagens mais sofisticadas – a busca tabu (Tabu T.) e o algoritmo genético – demonstraram resultados significativamente superiores. Ambas alcançaram uma média geral de desperdício em torno de 95,20. Contudo, o algoritmo genético se sobressaiu ao registrar 8 melhores resultados (em comparação aos 5 obtidos pela busca tabu) e apresentar um desvio padrão ligeiramente menor (101,67 versus 108,89), sugerindo maior robustez e estabilidade na obtenção de soluções de alta qualidade.

Assim, os resultados indicam que as heurísticas desenvolvidas são capazes de superar métodos tradicionais e estudos relacionados, sendo o algoritmo genético uma alternativa particularmente promissora para o problema em análise, ao combinar eficiência na redução do desperdício com uma variabilidade controlada, sem comprometer a aplicabilidade em instâncias mais complexas.

Capítulo 6

Conclusão

Este trabalho teve como objetivo comparar o desempenho das heurísticas propostas com métodos da literatura, incluindo um método guloso (Estudo 1: Poldi & Arenales [2003]) e algoritmos genéticos (Estudos 2: Boleta et al. [2005] e Estudo 3: Heis et al. [2006]). As análises foram conduzidas no contexto do problema de corte de estoque unidimensional inteiro, com foco na minimização do desperdício de material.

Os resultados indicaram que a heurística gulosa apresentou o pior desempenho, refletindo suas limitações em instâncias mais complexas. Em contraste, as heurísticas de busca local e busca tabu, desenvolvidas neste trabalho, demonstraram eficácia superior. Em particular, a busca tabu (Tabu T.) obteve uma média geral de desperdício de 97,41, registrando os melhores resultados em 5 classes e apresentando um desvio padrão de 108,89 – evidência de sua capacidade de explorar profundamente o espaço de soluções, embora com maior variabilidade.

Além disso, o algoritmo genético proposto, cuja segunda execução também alcançou uma média geral de 95,20, destacou-se por apresentar 8 melhores resultados e um desvio padrão levemente inferior (101,67), sugerindo maior robustez e estabilidade na obtenção de soluções de alta qualidade. Dessa forma, tanto a busca tabu quanto o algoritmo genético se mostraram capazes de competir com os métodos tradicionais, superando a heurística gulosa e aproximando-se dos melhores desempenhos apresentados pelos estudos da literatura.

Comparando com os métodos relacionados, embora os algoritmos genéticos dos Estudos 2 e 3 tenham se destacado em instâncias complexas, os resultados obtidos pelas heurísticas desenvolvidas neste trabalho demonstram que abordagens avançadas, especialmente o algoritmo genético proposto e a busca tabu, são alternativas promissoras para reduzir o desperdício de material. Tais achados reforçam a importância de explorar técnicas híbridas e adaptativas para enfrentar desafios crescentes em problemas de corte de estoque.

Por fim, a aplicação das heurísticas propostas em cenários reais constitui um passo relevante para validar sua eficácia e adaptabilidade. A utilização de dados reais e cenários industriais permitirá refinar as abordagens e torná-las mais competitivas em relação aos métodos consolidados, contribuindo para o avanço de soluções eficientes no problema de corte de estoque unidimensional inteiro e em desafios correlatos, com impacto positivo tanto na academia quanto na prática industrial.

Referências Bibliográficas

Aarts, E. & Lenstra, J. (2003). *Local Search in Combinatorial Optimization*. Princeton University Press.

Boleta, D. A. F.; de Araujo, S. A.; Constantino, A. A. & Poldi, K. C. (2005). Uma heurística para o problema de corte de estoque unidimensional inteiro. pp. 1870–1879.

Gau, G. & Wäscher, G. (1995). Cutgen: A problem generator for the standard one-dimensional cutting stock problem. *European Journal of Operational Research*, 84:572–579.

Gilmore, P. C. & Gomory, R. E. (1961). A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549.

Google (2023). Google colaboratory documentation. https://colab.research.google.com/notebooks/welcome.ipynb?hl=pt-BR. Accessed: 24-08-2024.

Gramani, M. C. N. (2008). Tomada de decisão no processo de cortagem: Minimizar a perda ou a troca de padrões de corte? (WPE: 155/2008).

Heis, A.; Constantino, A. A. & de Araujo, S. A. (2006). Um algoritmo genético para o problema de corte unidimensional inteiro. *Revista Brasileira de Pesquisa Operacional*, XX(XX):XX–XX.

Poldi, K. C. & Arenales, M. N. (2003). O Problema de Corte de Estoque Unidimensional Inteior Com Restrições de Estoque.

Python (2023). Python documentation. https://www.python.org/doc/. Accessed: 24-08-2024.

Santiago, P. H. R. (2016). O problema de corte de estoque unidimensional: Uma abordagem com geração de colunas. Dissertação de Mestrado.