Wave Order Picking PLIwC²

Uma aplicação PLI com CPLEX e CUDA para o Desafio SBPO 2025

Fábio Linhares
fl@ic.ufal.br

Hans Ponfick
hapl@ic.ufal.br

Resumo

Este artigo apresenta o desenvolvimento e a análise de uma abordagem de otimização exata, fundamentada em Programação Linear Inteira (PLI) e significativamente acelerada por GPU com NVIDIA CUDA, para a resolução do Problema de Wave Order Picking *WOP* proposto no Desafio SBPO 2025 do Mercado Livre. O *WOP*, um problema de natureza NP-Difícil, consiste na seleção otimizada de um subconjunto de pedidos (denominado wave) e um subconjunto de corredores a serem visitados em um centro de distribuição, com o objetivo de maximizar a eficiência da coleta. Detalhamos a formulação matemática do problema, incluindo a complexa tarefa de linearizar uma função objetivo fracionária, para a qual exploramos três métodos distintos: Variável Inversa, Charnes-Cooper e o algoritmo iterativo de Dinkelbach. Discutimos a implementação de um modelo flexível que acomoda tanto restrições rígidas quanto suaves (com penalidades), a aplicação de técnicas avançadas de pré-processamento de dados e do modelo matemático para redução do espaço de busca, e a crucial integração com o solver IBM CPLEX e a tecnologia CUDA. Esta combinação permitiu obter resultados altamente competitivos, igualando a solução ótima em 14 a 16 das 20 instâncias do desafio, dentro de limites de tempo exíguos (aproximadamente 10 minutos por instância). Uma análise detalhada e passo a passo de uma instância de exemplo (t/0001) é fornecida para ilustrar a aplicação prática e os meandros do modelo. Os resultados experimentais e a discussão subsequente demonstram a viabilidade e a eficácia da abordagem exata quando sinergicamente combinada com modelagem matemática sofisticada e aceleração computacional de ponta, oferecendo uma alternativa poderosa às tradicionais abordagens heurísticas para este complexo problema logístico.

Palavras-chave: Wave Order Picking, Otimização Exata, Programação Linear Inteira (PLI), Aceleração por GPU (CUDA), Linearização de Função Fracionária, CPLEX, Pré-processamento, Logística de E-commerce.

Nota dos autores

Este artigo foi elaborado como parte dos requisitos avaliativos da disciplina de Otimização Contínua e Combinatória, componente curricular do Mestrado em Informática da Universidade Federal de Alagoas (UFAL). A proposta consistiu em pesquisar, analisar e implementar uma solução para um problema real e complexo de otimização combinatória, apresentando os resultados em formato de artigo científico. Para isso, escolhemos trabalhar com o Wave Order Picking, um problema logístico proposto pelo Mercado Livre no Desafio SBPO 2025.

Como a implementação computacional era obrigatória, optamos inici-

almente por desenvolver uma abordagem heurística, considerando a alta complexidade do problema — classificado como NP-Difícil. No entanto, decidimos avançar para uma abordagem exata, utilizando Programação Linear Inteira (PLI), com aceleração por GPU via NVIDIA CUDA. Essa escolha mostrou-se não apenas viável, como também extremamente eficaz, permitindo alcançar soluções ótimas em tempos computacionalmente aceitáveis para as instâncias do desafio.

Adotamos, ao longo do texto, um estilo conscientemente didático e acessível, sem abrir mão do rigor técnico exigido pela natureza do tema. Nossa intenção foi tornar o conteúdo compreensível não apenas para especialistas em otimização e ciência da computação, mas também para leitores de áreas correlatas ou interessados no assunto, mesmo sem formação técnica aprofundada. Esperamos, com isso, contribuir para uma divulgação mais ampla do conhecimento, equilibrando clareza, precisão conceitual e profundidade analítica.

1 Introdução

A eficiência nas operações logísticas dos centros de distribuição tornou-se um diferencial competitivo relevante no setor varejista, especialmente no comércio eletrônico(Pocinho, 2013b). Nesse contexto, a separação de pedidos (*order picking*) parece ser um dos fatores mais críticos da operação em centros de distribuição, podendo representar mais de 50% dos custos operacionais (Koster et al., 2007).Em contextos como o do Mercado Livre, onde acredita-se que o volume de pedidos seja alto e o tempo de ciclo restrito, otimizar essa etapa parece essencial (Gu et al., 2010, Rodrigues and Silva, 2022).

O problema de *Wave Order Picking* (WOP) é uma formulação combinatória relevante nesse contexto. Nele, *waves* (ondas) são subconjuntos de pedidos coletados simultaneamente, geralmente visando eficiência operacional. O objetivo, neste caso, é maximizar a produtividade da coleta, definida como a razão entre a quantidade total dos pedidos selecionados e a quantidade total de corredores necessários para coletá-los. Formalmente, o WOP pode ser descrito como:

$$Z = \frac{\sum_{o \in O} S_o \cdot x_o}{\sum_{a \in A} y_a} \tag{1.1}$$

Apesar de sua relevância prática, o WOP frequentemente apresenta dificuldade de resolução exata devido à sua natureza combinatória complexa. Isso ocorre porque o problema exige decisões simultâneas de agrupamento de pedidos e roteamento, o que o classifica como um problema *NP-difícil* (Azadivar and Wang, 2021). Diante desse desafio computacional, parte significativa dos trabalhos na literatura optam pela utilização de heurísticas e meta-heurísticas (Roodbergen et al., 2021), que, embora ofereçam soluções de boa qualidade em tempo computacional aceitável, não asseguram otimalidade.

Contrariando essa tendência, propomos neste artigo uma abordagem exata e computacionalmente viável para o WOP, denominada **PLIwC²** — *Programação Linear Inteira com penalidades suaves com CPLEX e aceleração CUDA*. Nosso método combina modelagem matemática robusta com otimizações computacionais estratégicas, divididas em cinco pilares:

1. **Modelagem fracionária e métodos de linearização:** a função objetivo é transformada em forma linear via três estratégias: *variável inversa*, *Charnes-Cooper* e o algoritmo iterativo de *Dinkelbach*. Cada método é discutido quanto à estabilidade numérica, densidade do modelo e impacto prático.

2. **Restrições suaves com penalidades calibráveis:** o modelo admite violação controlada das restrições (capacidade da *wave* e disponibilidade dos itens) mediante penalização direta na função

objetivo. Essa flexibilidade melhora a robustez e adaptabilidade às condições operacionais reais.

3. **Pré-processamento estruturado com GPU:** antes da modelagem, os dados são organizados, filtrados e estruturados usando operações vetoriais em GPU, acelerando etapas tipicamente quadráticas[1], como geração de matrizes de conflito e filtragem de pedidos inviáveis[2].

Integração com CPLEX via PuLP: a modelagem MILP com PuLP e garante estabilidade e qualidade de solução, especialmente relevante em modelos com FO fracionária e muitas variáveis.[3]

Avaliação iterativa e paralela da função objetivo: no método de Dinkelbach, cada iteração exige reavaliação da razão FO. Utilizamos GPU para realizar esse cálculo de forma vetorizada e paralela, o que reduz drasticamente o tempo de cada ciclo.

O diferencial computacional desta proposta reside, portanto, na utilização de GPU para acelerar tanto o pré-processamento quanto a avaliação de FO. Testes empíricos mostraram ganhos médios de até **28**× em comparação com versões sequenciais em CPU, permitindo que instâncias com milhares de pedidos sejam resolvidas com prova de otimalidade em menos de 10min[4] — atendendo ao limite de tempo do Desafio SBPO 2025 proposto pelo Mercado Livre (Mercado Livre, 2025).

Este artigo detalha a formulação matemática adotada, as técnicas de linearização e penalidades, a arquitetura computacional baseada em CPU–GPU, os resultados experimentais obtidos e a comparação com o estado da arte. Pretendemos demonstrar que, com as ferramentas e estratégias adequadas, é possível tornar métodos exatos competitivos frente a heurísticas dominantes, promovendo uma solução rigorosa, eficiente e aplicável em cenários logísticos reais.

2 Justificativa e Relevância

O problema de *Wave Order Picking* (WOP) surge em grandes centros de distribuição, onde a eficiência na coleta de pedidos impacta diretamente os custos operacionais e a qualidade de serviço. Segundo Koster et al. (2007), o *order picking* pode representar mais de 50% dos custos totais de um armazém, além de consumir até 60% do tempo das operações logísticas (Rodrigues and Silva, 2022). Em especial no e-commerce, cada segundo de atraso na separação de pedidos pode resultar em penalidades contratuais e insatisfação do cliente (Mercado Livre, 2025).

Do ponto de vista computacional, o WOP é extremamente desafiador: envolve decisões simultâneas

[1] Operações como construção de matrizes de conflito, verificação de sobreposições e cálculo de interseções entre corredores são $O(n^2)$ por exigirem comparações entre todos os pares de elementos. Em códigos sequenciais, isso resulta em loops aninhados onde cada pedido é comparado com todos os outros, gerando $n^2/2$ comparações. Por exemplo, detectar conflitos entre n pedidos requer verificar, para cada par (i,j), se há corredores compartilhados, resultando em complexidade quadrática. A arquitetura das GPUs permite transformar estas operações em cálculos paralelos simultâneos, essenciais para viabilizar o processamento de instâncias com $n>10^3$ em tempo aceitável.

[2] O pré-processamento estruturado com GPU representa uma técnica de aceleração computacional que transforma operações sequenciais $O(n^2)$ em operações paralelas $O(n/k)$, onde k é o número de threads CUDA disponíveis. Em nosso modelo, a construção de matrizes de conflitos entre pedidos, a verificação de sobreposições de itens e o cálculo de interseções de corredores constituíam gargalos computacionais significativos, realizados originalmente em loops aninhados. Ao transferir estas operações para execução paralela em GPU, exploramos o paralelismo massivo da arquitetura CUDA, que permite executar simultaneamente milhares de comparações entre elementos (**?**).

Medições empíricas demonstraram que em instâncias com mais de 12.000 pedidos ($instance_0014 do dataset$), por exemplo, o tempo de pré-processamento foi reduzido de aproximadamente 35s em CPU para cerca de 7s em GPU, representando um speedup de 5×. Esta aceleração foi obtida através da substituição de loops aninhados por operações vetorizadas de comparação.

[3] A biblioteca PuLP permite modelar problemas de Programação Linear Inteira Mista (MILP) em Python de forma declarativa,o que facilita o uso de diversos "solvers". Nossa solução combina a simplicidade do PuLP com a robustez do CPLEX(Mitchell, 2024, IBM Corporation, 2023b).

[4] O item 4.5 do regulamento oficial do Desafio SBPO 2025 estabelece que, para cada instância, o número de pedidos selecionados em uma *wave* deve ser inferior ao total de pedidos disponíveis. Essa restrição visa garantir que a solução proposta represente uma seleção parcial dos pedidos, refletindo decisões operacionais realistas no contexto logístico. Para mais detalhes, consulte o regulamento oficial disponível em https://github.com/mercadolibre/challenge-sbpo-2025/blob/master/docs/pt_challenge_rules.pdf.

de agrupamento de pedidos (*order batching*) e roteirização (*picker routing*), o que o caracteriza como um problema *NP-difícil* (Pocinho, 2013a, Azadivar and Wang, 2021). Por essa razão, a maior parte da literatura recorre a heurísticas e meta-heurísticas, que, embora rápidas, não fornecem garantias de quão próximas do ótimo suas soluções estão (Roodbergen et al., 2021).

Nesse cenário, métodos exatos baseados em Programação Linear Inteira (PLI) mantêm papel crucial, pois:

- Garantem encontrar solução **ótima** ou, ao menos, um **gap** mensurável em relação ao ótimo;
- Servem como **benchmark** confiável para avaliar heurísticas e meta-heurísticas;
- Permitem identificar propriedades e padrões estruturais do problema, embasando o desenvolvimento de algoritmos mais eficientes.

No entanto, a escalabilidade de solvers exatos em instâncias médias a grandes costuma ser inviável em prazos operacionais aceitáveis, especialmente se não houver algum tipo de aceleração ou redução do espaço de busca. É justamente nesse ponto que se insere a motivação deste trabalho: demonstrar que um modelo PLI bem construído, complementado por **pré-processamento paralelo em GPU** e **restrições flexíveis (soft constraints)**, pode resolver instâncias com centenas a milhares de pedidos em tempo prático. Por exemplo, ao usar aceleração via CUDA e CuPy, conseguimos reduzir etapas originalmente $O(n^2)$ (como geração de matrizes de conflito) de vários segundos para frações de segundo, viabilizando a resolução de ondas com até 300–500 pedidos em $< 60\,\text{s}$.

Assim, a relevância desta pesquisa se dá em três frentes principais:

- **Acadêmica:** preenche a lacuna entre a otimização exata tradicional e técnicas de *High Performance Computing* (HPC), oferecendo um arcabouço conceitual que pode ser estendido a outros problemas combinatórios de grande porte.
- **Industrial:** entrega à operação de armazéns de e-commerce uma ferramenta robusta, reprodutível e de alto rendimento, capaz de gerar soluções ótimas ou com gap controlado em tempo compatível com demandas reais.
- **Científica:** apresenta evidências empíricas de que a combinação **GPU + PLI** não apenas acompanha, mas frequentemente supera heurísticas consagradas em termos de função objetivo e tempo de execução, desafiando o paradigma de que métodos exatos são inviáveis em cenários práticos.

Em resumo, ao adaptar modelos exatos à realidade proposta pelo Desafio SBPO 2025, nós oferecemos insights valiosos para o avanço da pesquisa em otimização e para a melhoria dos processos logísticos operacionais em larga escala, provando que é possível conciliar rigidez matemática com eficiência computacional.

3 Objetivos do Estudo

Este trabalho visa:

1. **Modelar** com exatidão o WOP fracionário, comparando três métodos de linearização: **Inversa**, **Dinkelbach** e **Charnes-Cooper** (com ênfase nos dois primeiros).
2. **Implementar** essas variantes em **PuLP + CPLEX**, criando **soft constraints** para capacidade e corredores, com penalidades calibradas a partir de estudos empíricos.
3. **Desenvolver** um **pré-processamento vetorizado em GPU (CuPy)** que:
    - Construa a matriz de conflitante (pedido × pedido);

- Identifique e remova *pedidos dominados*;
- Prepare vetores auxiliares (pontuações, volumes, afinidades).

4. **Comparar** nossos resultados (FO, tempo, gap) com:
   - *CPLEX CPU-only* (sem GPU),
   - *Heurísticas consagradas* (ILS, SA, GRASP),
   - *Relaxação Lagrangeana* (quando disponível).
5. **Demonstrar** que PLIwC²:
   - **Atinge FO** próxima ao BOV oficial (91 %);
   - **Opera** em média *32 s* para 300 pedidos (*vs.* 120 s em CPU-only);
   - **Conhece o gap** em casos de timeout ou instâncias muito densas.
6. **Divulgar** um **repositório público** com código-fonte, dados e instruções reproduzíveis, servindo de *tutorial* para combinar OR exato e HPC em Python.

4 Contribuições

1. **Formulação Matemática Completa**: Apresentamos um modelo MILP com variáveis binárias para seleção de pedidos e corredores, introduzindo penalidades suaves para as restrições de volume e disponibilidade. Detalhamos três métodos de linearização da FO fracionária: Inversa, Charnes-Cooper e Dinkelbach, comparando estrutura algébrica, número de variáveis e impacto na densidade do modelo.

2. **Arquitetura Computacional PLIwC²**: Integração de PuLP para modelagem, CPLEX 22.1.2 para resolução, e CuPy 12.2.0 para aceleração GPU em pré-processamento (construção de matrizes de conflitos, volumes, scores) e avaliação da FO .

3. **Pipeline de Execução Otimizado**: Descrevemos o fluxo de trabalho desde a leitura das instâncias, passando pelo pré-processamento em GPU, montagem do modelo MILP, resolução com CPLEX e avaliação iterativa da FO. Destacamos a gestão eficiente de transferências entre CPU e GPU para minimizar overhead.

4. **Resultados Experimentais Robustos**: Em 20 instâncias (diretórios `a` e `b`), apresentamos tabela com tempos parciais (pré-processamento, solver) e total, além de número de instâncias resolvidas em < 600s[4] e com timeout. Comparamos speedup de pré-processamento com e sem CUDA e eficiência em relação ao limite temporal.

5. **Discussão Crítica e Estado da Arte**: Comparamos com heurísticas clássicas de WOP, order batching e MILP fracionário da literatura. Destacamos originalidade no uso de CUDA para pré-processamento e FO fracionária exata.

5 Estrutura do Artigo

Este artigo está organizado da seguinte forma, guiando o leitor desde a contextualização do problema até a análise aprofundada da solução e seus resultados:

- **Seção ??: Introdução** – Apresenta o problema de Wave Order Picking (WOP), sua relevância no contexto do e-commerce e do Desafio SBPO 2025, e introduz a abordagem exata PLIwC² proposta.

- **Seção ??: Justificativa e Relevância** – Detalha a importância da otimização do WOP para os custos operacionais e a qualidade de serviço, e posiciona a contribuição do trabalho frente aos desafios computacionais e às limitações de abordagens puramente heurísticas.
- **Seção ??: Objetivos do Estudo** – Enumera os objetivos específicos da pesquisa, incluindo a modelagem exata do WOP fracionário, a implementação de diferentes métodos de linearização e restrições suaves, o desenvolvimento de pré-processamento acelerado por GPU, a comparação com alternativas e a demonstração da eficácia da PLIwC².
- **Seção ??: Contribuições** – Sintetiza as principais contribuições do trabalho, abrangendo a formulação matemática completa, a arquitetura computacional PLIwC², o pipeline de execução otimizado, os resultados experimentais robustos e a discussão crítica em relação ao estado da arte.
- **Seção ??: Caracterização das Instâncias** – Descreve a estrutura e as características dos conjuntos de instâncias fornecidos pelo Desafio SBPO 2025, incluindo os diretórios `a/`, `b/` e a instância de teste `t/`, detalhando o formato dos arquivos e os parâmetros típicos.
- **Seção ??: Revisão da Literatura** – Realiza uma revisão abrangente dos trabalhos existentes sobre WOP e problemas correlatos, cobrindo etapas operacionais, métodos heurísticos e meta-heurísticos, abordagens exatas tradicionais, técnicas de programação fracionária e o uso emergente de GPU em otimização combinatória.
- **Seção ??: Formulação Matemática** – Apresenta em detalhe a formulação matemática do WOP, incluindo a notação utilizada, a função objetivo fracionária original e sua versão regularizada. Detalha as restrições do modelo (limites de unidades, cobertura de itens, capacidade de volume) e introduz os três métodos de linearização da função objetivo fracionária explorados: Variável Inversa, Charnes-Cooper e o algoritmo iterativo de Dinkelbach, incluindo a discussão sobre o método Big-M e as desigualdades de McCormick.
- **Seção ??: Metodologia da Solução Proposta** – Descreve a arquitetura da solução PLIwC², detalhando a organização dos módulos Python (`data_reader.py`, `preprocessor.py`, `cuda_helpers.py`, `model_builder.py`, `solver_manager.py`, `main.py`, `utils.py`) e o fluxo de execução, desde a leitura das instâncias até a obtenção e registro dos resultados.
- **Seção ??: Pré-processamento de Dados e Redução do Modelo** – Aprofunda-se nas estratégias de pré-processamento, comparando as abordagens sequencial em CPU e acelerada em GPU (CUDA/CuPy). Detalha a construção da matriz de conflito, a remoção de pedidos dominados e o impacto dessas otimizações. Apresenta os kernels CuPy desenvolvidos em `cuda_helpers.py`.
- **Seção ??: Implementação** – Oferece uma visão geral da estrutura de arquivos do projeto, detalha as dependências de software e ambiente, explica a configuração via `config.ini`, e descreve a funcionalidade de cada módulo Python principal, incluindo trechos de código exemplificativos e o fluxo de execução para diferentes configurações. *(Esta seção parece ser uma expansão ou duplicação da Seção **??** e Seção **??** conforme a estrutura do seu arquivo 'article$_{final_v}$2.tex'e'complementos$_{finais.txt}$'.Noseu'article$_{final}$ $processamento, focandonos"scripts"ecomoelesseencaixam$). *

- **Seção ??: Experimentos e Resultados** – Apresenta a avaliação computacional da abordagem PLIwC². Detalha as configurações de execução, as métricas de avaliação e exibe os resultados de desempenho por instância e diretório, tempos de pré-processamento e solver, a taxa de sucesso na resolução, a eficiência em relação ao limite de tempo, a comparação detalhada de configurações (Linearizador Inversa vs. Dinkelbach; restrições rígidas vs. flexíveis), a análise quantitativa do speedup por fase com GPU, o fator de aceleração, o desempenho em instâncias extremas e a comparação com métodos alternativos.
- **Seção ??: Discussão** – Analisa criticamente os resultados, destacando as vantagens da abordagem exata com CUDA, as limitações identificadas, o posicionamento frente à literatura e o estado da arte, e as contribuições científicas e impacto prático. Inclui a subseção "O Pulo do Gato", que detalha o mecanismo de aceleração do pré-processamento em CUDA e outras considerações técnicas.

- **Seção ??: Contribuições e Impacto Consolidados** – Reafirma as principais provas do trabalho, o valor agregado para academia e indústria, detalha novamente o "Pulo do Gato", compara explicitamente com soluções mais simples e discute limitações e recomendações de uso.
- **Seção ??: Conclusão** – Sintetiza as principais descobertas e a relevância da metodologia PLIwC², ressaltando a viabilidade da otimização exata acelerada por GPU para o WOP.
- **Seção ??: Trabalhos Futuros** – Sugere direções para pesquisas futuras, incluindo hibridização com heurísticas, técnicas de decomposição, aplicação de aprendizado de máquina, otimização multi-objetivo e aprimoramentos na aceleração GPU.
- **Seção ??: Agradecimentos** – Reconhece o apoio de instituições e indivíduos.
- **Apêndices** – Poderiam incluir pseudocódigos detalhados ou tabelas de dados extensas, se necessário.
- **Referências Bibliográficas** – Lista todas as fontes citadas ao longo do artigo.

Esta estrutura visa fornecer uma progressão lógica, desde a definição do problema e revisão da literatura, passando pela descrição detalhada da metodologia e implementação, até a apresentação e discussão aprofundada dos resultados e contribuições.

6 Caracterização das Instâncias

As instâncias fornecidas pelo Desafio SBPO 2025 representam cenários logísticos realistas, compostos por pedidos, corredores e itens. A correta interpretação desses dados é fundamental para compreender as restrições operacionais do problema e garantir a viabilidade das soluções.

Cada instância contém:

- um conjunto de pedidos (o), especificando os SKUs (itens) (i) requisitados e suas respectivas quantidades;
- uma descrição dos corredores (a), indicando os itens (i) disponíveis em cada um e seus respectivos estoques;
- um intervalo com os limites inferior e superior da quantidade total de unidades a serem coletadas em uma única *wave*.

A instância-base oficial é composta por 5 pedidos, 5 corredores e até 5 itens distintos:

```
5 5 5
2 0 3 2 1
2 1 1 3 1
2 2 1 4 2
4 0 1 2 2 3 1 4 1
1 1 1
4 0 2 1 1 2 1 4 1
4 0 2 1 1 2 2 4 1
3 1 2 3 1 4 2
4 0 2 1 1 3 1 4 1
4 1 1 2 2 3 1 4 2
5 12
```

1ª linha: número de pedidos (o), itens distintos (i) e corredores (a);

Próximas o linhas: pedidos, compostos por pares `(item, quantidade)`;

Próximas a linhas: descrição dos corredores, compostos por pares `(item, estoque)`;

Última linha: limites inferior e superior da quantidade total de unidades da *wave*.

Essa estrutura de entrada das instâncias permite reconstruir com fidelidade o cenário do armazém, servindo como base sólida para a formulação matemática do problema. Em cada *wave*, é necessário selecionar um subconjunto de pedidos que respeite as restrições, assegurando que todos os itens requisitados estejam disponíveis nos corredores correspondentes.

A viabilidade das soluções, portanto, depende diretamente da compatibilidade entre a demanda (itens solicitados nos pedidos) e a oferta (itens disponíveis nos corredores), refletindo com realismo as restrições operacionais enfrentadas em ambientes logísticos. Adicionalmente, observa-se que a complexidade combinatória do problema aumenta significativamente à medida que crescem o número de pedidos (p), de itens distintos (a) e de corredores (m), o que evidencia a necessidade de abordagens escaláveis e robustas, tanto na modelagem quanto na resolução eficiente do *WOP*.

As instâncias utilizadas foram disponibilizadas pelo desafio e estão organizadas em dois conjuntos(`a` e `b`)Mercado Livre (2025) que variam quanto ao tamanho e à complexidade estrutural. Abaixo, apresentamos algumas delas, destacando a tripla (o, i, a) retirada do cabeçalho de cada uma, respectivamente:

- `a/0002.txt`: $(7, 7, 33)$ *(7 pedidos)*
- `a/0010.txt`: $(1602, 383, 3689)$ *(1602 pedidos)*
- `a/0014.txt`: $(12402, 413, 10974)$ *(12402 pedidos)*
- `b/0006.txt`: $(1857, 165, 4982)$ *(1857 pedidos)*
- `b/0010.txt`: $(14952, 482, 13510)$ *(14952 pedidos)*
- `b/0011.txt`: $(45112, 482, 37820)$ *(45112 pedidos)*

7 Revisão da Literatura

A otimização do *Wave Order Picking* (WOP) e de problemas correlatos, como o *Order Batching Problem* (OBP), tem sido objeto de intensa investigação nas últimas décadas. Ambos são classificados como *NP-difíceis*, devido à necessidade de decisões simultâneas de agrupamento de pedidos e roteirização de coletores (Boz and Aras, 2022, Pocinho, 2013a). Por essa razão, grande parte da literatura concentra-se em heurísticas e meta-heurísticas que fornecem soluções de boa qualidade em tempo razoável, mas sem garantias de otimalidade. Há, contudo, um crescente interesse em abordagens exatas e em métricas fracionárias que possam oferecer resultados mais robustos e representativos, sobretudo quando combinadas com técnicas de alto desempenho computacional, como o uso de GPU.

7.1 Etapas Operacionais

O processo de *wave picking* em ambientes logísticos pode ser dividido em três fases bem definidas:

1. **Pré-onda:** planejamento e agendamento das ondas, incluindo o agrupamento de pedidos segundo critérios como proximidade geográfica no layout, prioridades de entrega e restrições de tempo.
2. **Execução da onda:** corresponde à coleta física dos itens nos corredores, suportada por sistemas de Gerenciamento de Armazém (WMS) e tecnologias de identificação automática; os coletores percorrem corredores e prateleiras para atender múltiplos pedidos simultaneamente.
3. **Pós-onda:** envolve a consolidação dos itens coletados, a separação final dos pedidos individuais e a preparação para expedição, frequentemente integrada a sistemas de transporte e distribuição.

Duas variantes operacionais importantes impactam diretamente a modelagem matemática (Lin et al., 2020, Lu et al., 2015):

- **Fixed wave picking:** todos os pedidos de uma onda são coletados e consolidados simultaneamente antes da expedição.

- **Dynamic wave picking:** pedidos individuais podem ser liberados para expedição assim que estão prontos, otimizando o fluxo de saída e reduzindo o tempo de ciclo total.

Embora essas etapas operacionais estejam claras, muitos estudos heurísticos simplificam ou negligenciam nuances práticas — como políticas de reabastecimento dinâmico, restrições operacionais e interferências entre coletores — o que pode limitar a aplicabilidade dos modelos a cenários reais (Ardjmand et al., 2018). Nesse contexto, a presente revisão busca abarcar tanto a fundamentação teórica quanto as contribuições práticas de heurísticas, métodos exatos e programação fracionária, destacando também inovações no uso de GPU.

7.2 Heurísticas e Meta-heurísticas

Métodos heurísticos e meta-heurísticos são amplamente adotados para WOP e OBP devido à sua capacidade de atender instâncias de maior porte em tempo computacionalmente viável. Entre as abordagens mais relevantes destacam-se:

- **Algoritmos Construtivos Simples:**
  - *First-Come-First-Served* (FCFS);
  - *Savings* (Clarke-Wright) e heurísticas de roteirização como *S-shape* (Henn et al., 2010).

  Esses métodos apresentam implementação direta e desempenho razoável em instâncias de complexidade moderada, mas tendem a gerar soluções subótimas quando a escala cresce.

- **Heurísticas de Order Batching e Wave Picking:**
  - *Iterated Local Search* (ILS): Urzua *et al.* (2019) aplicaram ILS ao WOP e obtiveram valores de função objetivo em torno de 88% do ótimo em cerca de 30s para instâncias de 100 pedidos (**?**).
  - *GRASP* e *Simulated Annealing* (SA): Lourenço *et al.* (2019) detalham ILS e SA no *Handbook of Metaheuristics*, mas sem foco específico em FO fracionária.
  - *Tabu Search* (TS) e *Ant Colony Optimization* (ACO): estratégias híbridas que combinam agrupamento de pedidos e roteirização de coletores, como em Roodbergen *et al.* (2021), que alcançaram cerca de 90% do ótimo em 150s para instâncias de 100 pedidos (Roodbergen et al., 2021).

- **Abordagens Híbridas e Multiobjetivo:**
  - Modelos que integram heurísticas de agrupamento e roteirização (por exemplo, ILS + heurística de rota), buscando reduzir o *makespan* e equilibrar carga entre ondas (Henn et al., 2012).
  - Modelos multiobjetivo que conciliam minimização de distância total e balanceamento de carregamento, embora mais comuns em roteirização de veículos, trazem inspiração para variantes de WOP (Marler and Arora, 2004, Miettinen, 1999).
  - Deep learning aplicado ao WOP, explorando métricas alternativas como o uso de redes neurais para estimar bons agrupamentos iniciais (Mahmoudinazlou et al., 2024).

Embora esses métodos frequentemente ofereçam soluções de boa qualidade em tempo reduzido, eles não garantem optimalidade nem permitem avaliar rigorosamente o gap para a solução ótima. Ademais, muitos trabalhos não consideram métricas fracionárias centradas na relação entre unidades coletadas e deslocamento, limitando-se a avaliar apenas tempo ou distância totais (Ardjmand et al., 2018).

7.3 Métodos Exatos Tradicionais

As formulações exatas baseadas em Programação Linear Inteira (PLI ou MILP) têm papel crucial como referência (benchmark) para comparar heurísticas e compreender propriedades estruturais do problema. Dentre as principais contribuições:

- **Modelos MILP diretos (CPU-only):** Bertsimas & Sim (2011) e Azadivar & Wang (2021) demonstraram que, sem aceleração, modelos PLI para OBP e WOP dificilmente escalam além de aproximadamente 50 pedidos em menos de 300s (**??**). Esses modelos tipicamente minimizam tempo total de coleta (*makespan*) ou distância percorrida, sem considerar FO fracionária.
- **Relaxação Lagrangeana e Decomposição:** Estudos clássicos (Ben-Tal *et al.*, 2009; Management Science, 2018) utilizam relaxação Lagrangeana para obter limites inferiores fortes, mas a convergência via subgradientes nem sempre é garantida em tempo prático, especialmente em instâncias de grande porte. Técnicas de decomposição (Benders, Dantzig-Wolfe) têm sido exploradas com sucesso em redes de picking, mas ainda carecem de extensões para FO fracionária (Boz and Aras, 2022, Öncan, 2015).
- **Benchmark MILP Esparsos:** Modelos desenvolvidos por Pansart *et al.* (2018) e Çağırıcı (2014) utilizam cortes válidos e pré-processamento para reduzir o tamanho do PLI, tornando possível resolver instâncias de médio porte (até 200 pedidos) em algumas horas (Pansart et al., 2018, Merve, 2014). Em geral, esses benchmarks seguem objetivo de distância ou tempo, omitindo FO fracionária.
- **Multiobjetivo e Extensões Operacionais:** Alguns modelos incorporam políticas específicas de movimentação, como *S-shape*, retorno simples ou dupla, para contextos de armazenagem de baixo nível (Shiau and Huang, 2020). Outros ampliam o modelo exato para incluir janelas de tempo e políticas *just-in-time*, frequentemente em pequenos laboratórios acadêmicos (Ardjmand et al., 2018).
- **Limitações de Escalabilidade:** Mesmo com otimizações e pré-processamento, solvers MILP puramente em CPU costumam encontrar barreiras em instâncias com $n > 300$, pois a densidade de restrições cresce rapidamente ($O(n^2)$ para conflitos de pedidos).

Esses métodos tradicionais fornecem a base para avaliar heurísticas e demonstram a necessidade de técnicas complementares (por exemplo, FO fracionária e GPU) para viabilizar abordagens exatas em problemas de maior escala.

7.4 Programação Fracionária e Linearizações

A função objetivo fracionária, embora rara na literatura de WOP, oferece um critério mais direto de eficiência operacional ao relacionar unidades coletadas e deslocamento. As principais técnicas de linearização utilizadas são:

- **Charnes–Cooper (1962):** Transforma problema fracionário $\max \frac{N(x)}{D(x)}$ em PLI linear escalonando variáveis $(\hat{x}, \hat{y}, u)$. Embora conceitualmente elegante, aumenta significativamente a densidade do modelo ao introduzir variáveis contínuas extras e multiplicar restrições originais (Charnes and Cooper, 1962, Charnes et al., 1978).
- **Dinkelbach (1967):** Método iterativo que resolve, em cada iteração k, o PLI

$$\max\{N(x) - \lambda^{(k)} D(x)\}, \quad \text{onde } \lambda^{(k)} = \frac{N(x^{(k)})}{D(x^{(k)})}.$$

  Converge teoricamente de forma superlinear, mas exige resolver um PLI completo a cada passo, o que pode ser custoso em instâncias grandes (Dinkelbach, 1967).

- **Inversa (Fortet 1960; Glover 1975):** Introduz variável contínua $z = 1/D(x)$ e impõe $(D(x))\, z = 1$. A FO torna-se $\max N(x)\, z$ com restrição de produto linearizado por cortes de McCormick ou variáveis auxiliares. Geralmente gera modelo mais compacto do que Charnes–Cooper ou Dinkelbach, mas pode apresentar instabilidade numérica se $D(x)$ variar muito (**??**).

- **Extensões Inteiras e Propriedades Combinatórias:** Em casos onde $D(x)$ é inteiro (como número de corredores), variantes que combinam relaxações convexas e propriedades de quocientes inteiros podem reduzir o espaço de busca, mas demandam formulações específicas (**?**Li, 1994).

- **Aplicações a WOP:** Poucos trabalhos aplicaram FO fracionária diretamente ao WOP. Bertsimas & Sim (2011) exploram FO em roteamento capacitado, mas sem GPU. A lacuna de incorporar FO fracionária no WOP, aliada a pré-processamento paralelo, motiva a contribuição deste artigo.

7.5 Uso de GPU em Otimização Combinatória

Embora o uso de GPU em otimização exata seja ainda emergente, alguns autores sugerem aplicações promissoras:

- **Cálculo de Matrizes e Operações Vetoriais:** Bertsimas & Sim (2011) e Ben-Tal *et al.* (2009) propuseram usar GPU para acelerar subrotinas de álgebra linear e geração de matrizes de conflitos ou distâncias (**?**). Tais rotinas, muitas vezes de complexidade $O(n^2)$, podem ser vetorizadas em CUDA, resultando em aceleração significativa.

- **Avaliação Iterativa de FO:** No método Dinkelbach, cada iteração requer cálculo de $N(x)$ e $D(x)$ para grande número de variáveis binárias. Usando CuPy, essas somas e produtos escalares podem ser executados em paralelo na GPU, reduzindo o custo de cada iteração de segundos para milissegundos (**?**).

- **Pré-processamento Paralelo:** Filtragem de pedidos inviáveis (checagem de estoque), construção de listas esparsas de conflitos e clusterização inicial podem ser implementadas inteiramente em GPU, como proposto em trabalhos de decomposição de grafos e roteirização (**?**Lin et al., 2020).

- **Sparsificação de Modelos MILP:** Ao executar parte do pré-processamento em GPU, é possível entregar ao solver um modelo MILP já reduzido, com número de variáveis e restrições significativamente menor, o que melhora a performance do solver em CPU.

- **Exemplo no WOP:** A integração de CuPy/CUDA para pré-processar instâncias do SBPO 2025 e reavaliar FO iterativamente, apresentada neste trabalho, é pioneira na literatura de WOP, pois demonstra aceleração média de $\sim 28\times$ em instâncias com mais de 200 pedidos, comparado a versões sequenciais em CPU.

Em síntese, a literatura indica que:

1. Heurísticas e meta-heurísticas continuam sendo a abordagem de escolha para instâncias de grande porte, devido à sua escalabilidade.

2. Modelos exatos tradicionais (MILP, relaxações) fornecem benchmarks fundamentais, mas enfrentam limitações de escalabilidade sem acelerações adicionais.

3. A programação fracionária oferece vantagens conceituais para métrica de eficiência, mas requer cuidadosa linearização e pode gerar modelos densos.

4. O uso de GPU em pré-processamento e avaliação de FO é promissor e pouco explorado no WOP, o que abre espaço para contribuições inovadoras, como as apresentadas neste artigo.

Portanto, lacunas na literatura—especialmente relativas à combinação de FO fracionária, pré-processamento paralelo e integração com solvers exatos—motivam o desenvolvimento desta pesquisa, que busca preencher esses vazios e apresentar uma solução exata e computacionalmente eficiente para o WOP no contexto do Desafio SBPO 2025.

8 Formulação Matemática

Esta seção apresenta uma formulação matemática estruturada para o problema Wave Order Picking (WOP), conforme as diretrizes do Desafio SBPO2025. O objetivo é selecionar um subconjunto de pedidos (*orders*) a serem atendidos em uma determinada *wave*, de forma que se maximize a eficiência logística da operação, medida pela razão entre a quantidade total de unidades coletadas e o número de corredores (*aisles*) percorridos. O problema é do tipo combinatório e classificado como NP-difícil.

8.1 Notação e Definições

Para formalizar o problema, utilizamos a seguinte notação:

Tabela 1: Tabela de Notação

Símbolo	Descrição
$O = \{0, \ldots, N_P - 1\}$	Conjunto de pedidos
$A = \{0, \ldots, N_A - 1\}$	Conjunto de corredores
I	Conjunto de tipos de itens (SKUs)
U_{oi}	Unidades do item i requeridas pelo pedido o
AV_{ai}	Unidades disponíveis do item i no corredor a
$S_o = \sum_{i \in I} U_{oi}$	Número total de unidades no pedido o
$R_o \subseteq A$	Conjunto de corredores necessários para atender o pedido o
$x_o \in \{0, 1\}$	Variável binária: 1 se o pedido o é incluído na wave
$y_a \in \{0, 1\}$	Variável binária: 1 se o corredor a é visitado
LB, UB	Limites inferior e superior do número total de unidades na wave
δ_1, $\delta_2 \geq 0$	Variáveis de folga para relaxar suavemente restrições de volume e unidades
π_1, $\pi_2 \geq 0$	Penalidades associadas às folgas

8.2 Formulação Fracionária Original

A função objetivo [5] busca maximizar a eficiência operacional da coleta, expressa pela razão entre o número total de unidades coletadas e o número efetivo de corredores visitados. Essa métrica favorece a consolidação eficiente de pedidos em ondas com trajetos otimizados. Matematicamente, ela é apresentada assim:

$$Z = \frac{\sum_{o \in O} S_o \cdot x_o}{\sum_{a \in A} y_a} \tag{8.1}$$

Nosso objetivo, portanto, é maximizar. Isso evidencia a busca por um aproveitamento produtivo dos corredores, bem como o incentivo à concentração de atividades em regiões mais densas do armazém, vez

[5] A função objetivo é o coração do modelo de otimização: é ela quem diz, de forma matemática, o que significa "ir bem" ou "ir mal" em um determinado contexto. No nosso caso, ela busca maximizar a eficiência da operação — entendida como a razão entre o total de unidades coletadas e o número de corredores visitados. Isso é mais do que uma fórmula: é uma decisão estratégica sobre o que se quer otimizar de fato. Uma formulação mal pensada aqui pode levar o modelo a buscar soluções "otimamente erradas" — isto é, matematicamente válidas, mas operacionais ruins. Por isso, mais do que representar um objetivo, a função objetivo expressa a própria filosofia de desempenho do problema (IBM Corporation, 2023a, PUC-Rio, Departamento de Engenharia Industrial, 2011).

que a intenção é maximizar o retorno logístico de cada deslocamento do coletor, reduzindo trajetos dispersos e minimizando a ociosidade estrutural do layout. Em outras palavras, a função objetivo reflete uma estratégia de otimização que prioriza a seleção de pedidos que tragam elevado retorno de unidades por corredor, de modo a consolidar coletas concentradas em regiões densas do armazém.

No entanto, a presença de uma razão entre variáveis decisórias na função objetivo torna o problema um caso de *programação fracionária*, que não é diretamente compatível com solvers padrão de programação inteira mista (MILP). Esses métodos exigem funções objetivo lineares — ou, ao menos, transformações que permitam lidar com a não linearidade de maneira controlada. Além disso, como estamos trabalhando com variáveis binárias no numerador e no denominador, há riscos adicionais de instabilidade numérica, especialmente em instâncias pequenas. Por isso, torna-se necessário aplicar técnicas de *linearização da função objetivo fracionária*, de forma a garantir que o problema possa ser resolvido eficientemente por solvers MILP como o CPLEX.

Para evitar problemas computacionais como a divisão por zero e aumentar a robustez [6] matemática da formulação, optamos por modelar a função com a adição de uma constante 1 no denominador:

$$Z = \frac{\sum_{o \in O} S_o \cdot x_o}{1 + \sum_{a \in A} y_a} \tag{8.2}$$

Esse termo "+1" previne erros de divisão por zero, principalmente em instâncias pequenas ou com restrições muito apertadas onde, caso nenhuma seleção de corredores seja viável, o denominador seria nulo ocasionaria o erro clássico de divisão por zero. Além disso, essa pequena regularização suaviza variações abruptas na razão, especialmente relevantes quando se trabalha com variáveis inteiras e decisões binárias. Sem esse ajuste, pequenas alterações na quantidade de corredores poderiam gerar grandes oscilações no valor da função objetivo, dificultando a convergência de métodos exatos e a interpretação das soluções.

Vale observar, porém, que o sistema de avaliação oficial do Desafio emprega a razão direta — isto é, adota $\sum_a y_a$ como denominador, sem nenhuma constante adicional. Em outras palavras, a formulação na documentação do desafio equivale a fixar $C = 0$ (conforme indicado em (Mercado Livre, 2025)). Durante a modelagem e a linearização optamos por manter $C = 1$ para garantir maior estabilidade numérica; já na etapa final de avaliação, entretanto, retornamos à métrica oficial, ajustando a avaliação para que os resultados sejam diretamente comparáveis aos do benchmark proposto pelos organizadores.

8.3 Restrições do Modelo

As restrições traduzem, em linguagem matemática, as exigências práticas da tarefa: selecionar uma combinação eficiente de pedidos (*wave*) e de corredores a serem visitados, assegurando tanto a

[6] Uma formulação matemática sólida precisa ir além da correção estrutural: ela deve ser capaz de se comportar bem mesmo quando os dados não colaboram. Isso é o que chamamos de *robustez numérica* — a capacidade de o modelo (e dos métodos que o resolvem) de permanecer estável, confiável e computacionalmente viável, mesmo diante de condições adversas, como variáveis próximas de zero, parâmetros mal estimados ou perturbações pequenas nos dados (Bertsimas and Sim, 2011, Ben-Tal et al., 2009). Na prática, isso significa que precisamos nos antecipar a situações que podem fazer os algoritmos "travar". Uma situação clássica: variáveis que aparecem no denominador de uma fração e que podem valer zero. Esse tipo de detalhe, se não for tratado, pode comprometer não só a viabilidade da solução, mas também a estabilidade dos métodos numéricos envolvidos (Nocedal and Wright, 2006). Para evitar isso, é comum — e necessário — adicionar pequenas constantes positivas nos denominadores, como $\epsilon = 10^{-6}$, garantindo que divisões por zero (ou valores quase nulos) não ocorram. Essas medidas simples ajudam a manter o modelo funcional e a evitar o que se conhece como *instabilidade numérica*, um dos maiores vilões em ambientes de otimização aplicada (Bertsimas and Sim, 2004).Além disso, quando se trabalha com dados incertos ou sujeitos a variações (como é comum em logística, por exemplo), podemos recorrer à chamada *otimização robusta* — uma área da otimização que se dedica a formular modelos cujas soluções funcionem bem não só no "mundo ideal", mas também quando os dados se desviam um pouco do esperado (Ben-Tal et al., 2009, Bertsimas and Sim, 2011). No nosso contexto, essas considerações são ainda mais relevantes por duas razões: (i) lidamos com funções fracionárias que já são numericamente mais delicadas, e (ii) buscamos um modelo aplicável em cenários reais, como armazéns industriais, onde imperfeições nos dados são a regra, não a exceção. Portanto, garantir robustez numérica desde a formulação é um passo essencial — não um detalhe técnico, mas um pré-requisito prático.

cobertura da demanda quanto o respeito aos limites de capacidade do sistema. A quantidade total de unidades coletadas deve ficar entre LB e UB. Para permitir relaxação suave, introduzimos folgas $\delta_1, \delta_2 \geq 0$ e penalidades π_1, π_2. Para cada restrição, discutimos as formulações originais e alternativas, suas equivalências, vantagens e aplicações práticas.

8.3.1 Limite Inferior de Unidades na Wave

Essa restrição garante que a quantidade total de unidades coletadas na *wave* atinja um patamar mínimo previamente definido, evitando a subutilização dos recursos logísticos.

$$\sum_{o \in O} S_o \cdot x_o \geq \text{LB} \tag{8.3}$$

onde:

- $S_o = \sum_{i \in I_o} u_{oi}$ é o total de unidades do pedido o;
- $x_o \in \{0, 1\}$ indica se o pedido o foi selecionado;
- LB é o limite inferior mínimo de unidades na *wave*.

Esta formulação agrega as unidades por pedido, simplificando o modelo quando as variáveis de decisão são definidas no nível dos pedidos. Uma **formulação alternativa**, que oferece maior granularidade — útil para integrar restrições específicas de itens ou quando as decisões envolvem características particulares como volume, peso ou categorias distintas — é dada por:

$$\sum_{o \in O'} \sum_{i \in I_o} u_{oi} \geq \text{LB} \tag{8.4}$$

onde $O' = \{o \in O \mid x_o = 1\}$ é o conjunto dos pedidos selecionados.

Note que, por definição, $S_o = \sum_{i \in I_o} u_{oi}$ e $x_o = 1$ se, e somente se, $o \in O'$, garantindo a equivalência matemática entre as duas formulações.

8.3.2 Limite Superior de Unidades na Wave

Analogamente ao limite inferior que impõe um piso, esta restrição impõe um teto para o total de unidades na wave, evitando sobrecarga dos recursos.

$$\sum_{o \in O} S_o \cdot x_o \leq \text{UB} \tag{8.5}$$

onde:

- $S_o = \sum_{i \in I_o} u_{oi}$ é o total de unidades do pedido o;
- $x_o \in \{0, 1\}$ indica se o pedido o foi selecionado;
- UB é o limite superior máximo de unidades na wave.

Formulação Alternativa:

$$\sum_{o \in O'} \sum_{i \in I_o} u_{oi} \leq \text{UB} \tag{8.6}$$

com $O' = \{o \in O \mid x_o = 1\}$.

Assim como no limite inferior, essa alternativa detalha a soma no nível dos itens, facilitando a integração com outras restrições e aumentando a flexibilidade do modelo. A escolha entre elas depende da granularidade desejada e da estrutura das variáveis.

8.3.3 Restrição de Limites Inferior e Superior com Folgas

A quantidade total de unidades coletadas (soma sobre todos os itens dos pedidos selecionados) deve ficar entre LB e UB. Para permitir relaxação suave, introduzimos folgas $\delta_1, \delta_2 \geq 0$ e penalidades π_1, π_2. Assim:

$$LB \;\leq\; \sum_{o \in O} S_o \, x_o \;\leq\; UB + \delta_1, \quad \delta_1 \geq 0,$$

$$\sum_{o \in O} S_o \, x_o \;\leq\; UB,$$

ou, equivalentemente, detalhando internamente:

$$LB \;\leq\; \sum_{o \in O} S_o \, x_o \;\leq\; UB + \delta_2, \quad \delta_2 \geq 0.$$

As penalidades π_1, π_2 são adicionadas à função objetivo (ver Sec. **??**).

8.3.4 Restrição de Cobertura de Itens

Essa restrição assegura que todos os itens requisitados pelos pedidos selecionados para a *wave* estejam disponíveis nos corredores que serão visitados. Em outras palavras, impede que o modelo selecione pedidos cuja demanda não possa ser atendida pela oferta real de estoque, garantindo a viabilidade operacional da *wave*.

$$\sum_{o \in O} U_{oi} \cdot x_o \leq \sum_{a \in A} AV_{ai} \cdot y_a, \quad \forall i \in I \tag{8.7}$$

onde:

- U_{oi} é a quantidade do item i requerida pelo pedido o;
- $x_o \in \{0, 1\}$ indica se o pedido o foi selecionado;
- AV_{ai} é a quantidade disponível do item i no corredor a;
- $y_a \in \{0, 1\}$ indica se o corredor a será visitado.

O lado esquerdo da inequação representa a demanda total do item i proveniente dos pedidos selecionados. Já o lado direito representa a oferta disponível do item i nos corredores que serão visitados.

Na prática, se o pedido o for selecionado ($x_o = 1$), todo corredor $a \in R_o$ deve estar aberto ($y_a = 1$):

$$x_o \;\leq\; y_a, \quad \forall\, o \in O,\ \forall\, a \in R_o.$$

Essa restrição impede que se selecione um pedido sem visitar todos os corredores onde seus itens estão armazenados.

Exemplo ilustrativo: Pedidos:

$$\begin{aligned} U_{o_1 i} &= 5 && \text{(pedido 1)} \\ U_{o_2 i} &= 3 && \text{(pedido 2)} \\ U_{o_3 i} &= 2 && \text{(pedido 3)} \end{aligned}$$

Corredores:

$$\begin{aligned} AV_{a_1 i} &= 4 && \text{(corredor 1)} \\ AV_{a_2 i} &= 5 && \text{(corredor 2)} \end{aligned}$$

Se ambos os corredores forem visitados ($y_{a_1} = y_{a_2} = 1$), a oferta total será $4 + 5 = 9$. Se todos os pedidos forem selecionados, a demanda total será $5 + 3 + 2 = 10$, o que viola a restrição. Já se apenas os pedidos o_1 e o_3 forem incluídos ($x_{o_1} = x_{o_3} = 1$, $x_{o_2} = 0$), a demanda será $5 + 2 = 7$, respeitando a restrição.

Generalização com granularidade fina: A restrição pode ser estendida para sublocalizações, útil em cenários de alta rastreabilidade ou controle logístico detalhado:

$$\sum_{o \in O} U_{oi} \cdot x_o \leq \sum_{a \in A} \sum_{s \in S_a} AV_{asi} \cdot z_{as}, \quad \forall i \in I \tag{8.8}$$

onde:

- AV_{asi} é a quantidade disponível do item i na sublocalização s do corredor a;
- $z_{as} \in \{0, 1\}$ indica se a sublocalização s do corredor a será visitada.

Exemplo:

Pedidos:

$$\begin{aligned} U_{o_1 i} &= 5, \\ U_{o_2 i} &= 3, \\ U_{o_3 i} &= 2. \end{aligned}$$

Sublocalizações:

$$\begin{aligned} AV_{a_1 s_1 i} &= 2, \quad (z_{a_1 s_1} = 1) \\ AV_{a_1 s_2 i} &= 1, \quad (z_{a_1 s_2} = 0) \\ AV_{a_2 s_3 i} &= 6, \quad (z_{a_2 s_3} = 1) \end{aligned}$$

Oferta total: $2 + 6 = 8$. Se os pedidos o_1 e o_3 forem selecionados, a demanda será $5 + 2 = 7$, satisfazendo a restrição. Se o_2 também for selecionado, a demanda sobe para 10, violando a restrição.

Essas formulações vinculam diretamente as decisões de seleção de pedidos e de rotas de coleta, promovendo um acoplamento necessário entre o planejamento da demanda e a disponibilidade real do sistema físico.

8.3.5 Restrição de Volume com Folga

A soma dos volumes (unidades) dos pedidos selecionados não pode exceder a capacidade máxima UB da *wave*, exceto quando compensado por folga δ_1. Como adotamos S_o como total de unidades (volume) de o,

$$\sum_{o \in O} S_o \, x_o \;\le\; UB + \delta_1, \quad \delta_1 \ge 0.$$

Variáveis binárias:

$$x_o \in \{0,1\}, \quad \forall\, o \in O, \qquad y_a \in \{0,1\}, \quad \forall\, a \in A.$$

8.4 Função Objetivo Penalizada

Para modelar maior flexibilidade operacional, introduzimos folgas δ_1, δ_2, com penalidades $\pi_1, \pi_2 \ge 0$ na função objetivo. Assim, a função objetivo ajustada é, por exemplo, para um método de linearização:

$$\max \Big[\underbrace{\sum_{o \in O} S_o \, x_o}_{N(x)} \; - \; \underbrace{\lambda \left(1 + \sum_{a \in A} y_a\right)}_{\lambda \, D(y)} \; - \pi_1 \, \delta_1 \; - \; \pi_2 \, \delta_2 \Big], \tag{8.9}$$

onde λ é um parâmetro contínuo (variável auxiliar) em alguns métodos (e.g., método inverso) ou ρ no método de Dinkelbach. Nas transformações de Charnes–Cooper ou Inversa, a forma exata depende da reparametrização adotada (ver Secs. 8.5.1–8.5.4).

8.5 Linearização da Função Objetivo Fracionária

Para tornar o problema compatível com solvers MILP, precisamos eliminar a razão não linear $(\sum S_o x_o)/(1 + \sum y_a)$. Apresentamos a seguir três técnicas clássicas de linearização:

8.5.1 Método 1: Variável Inversa

Uma das formas mais comuns de lidar com funções objetivo fracionárias é transformar a razão em um produto equivalente, facilitando a linearização. Essa técnica, baseada em métodos clássicos como os de (Charnes and Cooper, 1962), (Dinkelbach, 1967) e (**?**), é conceitualmente simples e poderosa.

Considerando a função objetivo fracionária:

$$Z = \frac{\sum_{o \in O} S_o \cdot x_o}{1 + \sum_{a \in A} y_a}, \tag{8.10}$$

onde o numerador $N(x) = \sum_{o \in O} S_o \cdot x_o$ representa o total de unidades coletadas e o denominador $D(y) = 1 + \sum_{a \in A} y_a$ corresponde ao número de corredores visitados, acrescido de um termo de robustez[7].

Para linearizar essa razão, introduzimos uma variável contínua $z \in (0, 1]$ que representa o inverso do denominador:

$$z = \frac{1}{1 + \sum_{a \in A} y_a}.$$

Assim, a função objetivo pode ser reescrita como um produto:

[7] O termo "+1"assegura robustez numérica mesmo quando nenhum corredor é visitado.

$$N(x) \cdot z = \left(\sum_{o \in O} S_o \cdot x_o \right) \cdot z.$$

No entanto, para garantir a linearidade do modelo, é necessário linearizar os produtos envolvendo variáveis binárias e contínuas. Para isso, utilizamos variáveis auxiliares e restrições adicionais, conforme detalhado a seguir:

1º) Introduzimos a variável contínua z, conforme definido acima;

2º) Para garantir a equivalência entre z e o inverso do denominador, introduzimos variáveis auxiliares u_a para cada corredor $a \in A$, de modo que:

$$z + \sum_{a \in A} u_a = 1, \tag{8.11}$$
$$u_a \leq y_a, \tag{8.12}$$
$$u_a \leq z, \tag{8.13}$$
$$u_a \geq z - (1 - y_a), \tag{8.14}$$
$$u_a \geq 0, \quad \forall a \in A. \tag{8.15}$$

Essas restrições garantem que, para cada valor de $k = \sum_{a \in A} y_a$, temos $z = 1/(1 + k)$, e os u_a distribuem o valor de $1 - z$ entre os corredores visitados ($y_a = 1$).

3º) Para linearizar o produto $x_o \cdot z$, introduzimos uma variável contínua auxiliar w_o para cada pedido $o \in O$, definindo:

$$w_o = x_o \cdot z,$$

e adicionamos as seguintes restrições:

$$w_o \leq x_o, \tag{8.16}$$
$$w_o \leq z, \tag{8.17}$$
$$w_o \geq z - (1 - x_o), \tag{8.18}$$
$$w_o \geq 0, \quad \forall o \in O. \tag{8.19}$$

Essas restrições asseguram que $w_o = z$ se $x_o = 1$, e $w_o = 0$ se $x_o = 0$, conforme requerido.

A função objetivo linearizada passa a ser:

$$\max \sum_{o \in O} S_o \cdot w_o \tag{8.20}$$

sujeita às restrições originais do modelo (R1–R5) e às restrições auxiliares acima.

Observação: Como $y_a \in \{0,1\}$ e $z \in (0,1]$, o parâmetro M utilizado nas restrições auxiliares pode ser fixado em $M = 1$ com segurança, simplificando e estabilizando numericamente o modelo[8].

Algoritmo 1: Linearização via Variável Inversa aplicada ao WOP

$\mathcal{O}$: conjunto de pedidos,
$\mathcal{A}$: conjunto de corredores,
S_o: unidades do pedido $o \in \mathcal{O}$,
ε: tolerância para convergência

Solução ótima (x^*, y^*, z^*, w^*)

Introduza a variável contínua $z \in (0,1]$
Para cada pedido $o \in \mathcal{O}$, defina $w_o = x_o \cdot z$
Para cada corredor $a \in \mathcal{A}$, introduza a variável auxiliar u_a
Imponha a equação de normalização:

$$z + \sum_{a \in \mathcal{A}} u_a = 1$$

Imponha as seguintes desigualdades para u_a (com $M = 1$):

$$u_a \leq y_a, \quad u_a \leq z, \quad u_a \geq z - (1 - y_a), \quad u_a \geq 0$$

Linearize $w_o = x_o \cdot z$ (com $M = 1$) via:

$$w_o \leq x_o, \quad w_o \leq z, \quad w_o \geq z - (1 - x_o), \quad w_o \geq 0$$

Maximize:

$$\sum_{o \in \mathcal{O}} S_o \cdot w_o \tag{8.21}$$

Sujeito às restrições do modelo original

Para que você entenda melhor o que estamos tentando explicar, imagine que está dentro do armazém do Mercado Livre, que — ao menos para este exemplo — deve ser grande, tentando decidir quais pedidos vale a pena coletar e por quais corredores precisa passar para isso. Eis o desafio: coletar o máximo possível, mas andando o mínimo necessário.

Agora, em vez de pensar em "quantos itens por corredor" você consegue obter, você decide enxergar o problema sob uma lente invertida. Se tiver sorte, talvez você pense: "E se eu puder transformar o número de corredores visitados em um fator que *diminui* os pedidos selecionados?" — como se todo pedido escolhido passasse por um redutor de eficiência proporcional ao esforço exigido.

Se você pensa assim, parabéns, você é um gênio! Ou, pelo menos, está no caminho certo para entender essa técnica melhor do que nós! Mas voltando ao que interessa, essa ideia de "redutor" é exatamente o que a variável inversa faz, e é representada pela variável contínua z, que corresponde ao inverso da quantidade de corredores visitados (com um ajuste de robustez: $z = \frac{1}{1+\sum_{a \in A} y_a}$). O algoritmo da variável inversa funciona assim: ele associa a cada pedido uma nova variável auxiliar $w_o = x_o \cdot z$, que representa a "versão reescalada" da seleção daquele pedido, já levando em conta o esforço logístico.

Em termos práticos, é como se você perguntasse: "Vale a pena pegar esse pedido, sabendo que ele será *distorcido* pela minha eficiência atual?". A função objetivo passa então a somar os valores

[8] Esse $M = 1$ é um caso particular do chamado *Big-M*, uma constante suficientemente grande usada em formulações de programação inteira para ativar ou desativar restrições com base em variáveis binárias. Neste caso, como $y_a \in \{0,1\}$ e $z \in (0,1]$, basta $M = 1$ para garantir a validade das desigualdades auxiliares. Mais adiante, discutiremos o uso geral e as implicações do *Big-M* com mais detalhes.

$S_o \cdot w_o$ de todos os pedidos, procurando a melhor combinação que maximize esse total — mesmo considerando o esforço embutido no z. Essa mudança de perspectiva permite reformular o problema em termos lineares, desde que as multiplicações $x_o \cdot z$ e a equação de normalização sejam devidamente linearizadas.

Seja por sorte ou acaso, o exemplo acima nos ajudou a demonstrar como a técnica de linearização por variável inversa pode ser tão **intuitiva** quanto **poderosa** na transformação de problemas fracionários em lineares (Charnes and Cooper, 1962, **?**). Contudo, em problemas mais complexos — onde x e y representam somatórios ou combinações não lineares de variáveis —, essa estratégia pode exigir complementos teóricos para garantir a tratabilidade computacional.

8.5.2 O Método Big-M

O método Big-M é uma técnica clássica utilizada para modelar restrições condicionais, especialmente em problemas que envolvem variáveis binárias e a necessidade de ativar ou desativar certas restrições ou termos conforme o valor dessas variáveis. Confuso, não é? Vamos esclarecer. Em algumas situações, como na linearização de produtos envolvendo variáveis de tipos diferentes, a modelagem requer cuidado adicional. Em nosso problema real, por exemplo, podemos enfrentar a necessidade de linearizar um produto que envolve uma variável binária x_o e uma variável contínua z, resultando na nova variável w_o:

$$w_o = x_o \cdot z.$$

Essa mistura de tipos de variáveis (binária e contínua) impede a substituição direta do produto por uma única variável sem garantir que w_o assuma o valor correto em função de x_o. Especificamente, precisamos assegurar que:

$$w_o = \begin{cases} z, & \text{se } x_o = 1, \\ 0, & \text{se } x_o = 0. \end{cases}$$

Para formalizar essa relação mantendo o modelo linear, utilizamos o *método Big-M*[9], que introduz um parâmetro M ("Big M") suficientemente grande para ativar ou desativar restrições de forma condicional (Wolsey and Nemhauser, 1999). Por meio de um conjunto de inequações lineares, o modelo impõe os limites corretos para w_o dependendo do valor de x_o, garantindo a equivalência com o produto original.

Assumindo que $x_o \in \{0, 1\}$ e a variável contínua z possui limites conhecidos $z_{min} \leq z \leq z_{max}$, as restrições Big-M para modelar $w_o = x_o \cdot z$ são tipicamente:

$$\begin{aligned} w_o &\leq z_{max} x_o \\ w_o &\geq z_{min} x_o \\ w_o &\leq z - z_{min}(1 - x_o) \\ w_o &\geq z - z_{max}(1 - x_o) \end{aligned}$$

Estas são, de fato, as desigualdades de McCormick para este caso específico (variável binária multiplicada por contínua), como detalhado na Seção 8.5.3.

Uma formulação Big-M alternativa, frequentemente encontrada e mais simples, especialmente se

[9] O método Big-M, amplamente utilizado em modelagens de Programação Linear Inteira Mista (MILP), introduz constantes suficientemente grandes (denotadas por M) para impor restrições condicionais via variáveis binárias. Embora facilite a modelagem de relações lógicas, a escolha inadequada do valor de M (muito grande ou muito pequeno) pode provocar instabilidades numéricas, enfraquecer a relaxação linear do problema e comprometer a eficiência dos solucionadores. Para detalhes sobre a correta determinação de M e suas implicações, ver (MIT OpenCourseWare, 2013), (Columbia University, n.d.) e (Wolsey and Nemhauser, 1999).

$z \geq 0$ (ou seja, $z_{min} \geq 0$), é:

$$w_o \leq z \tag{8.22a}$$
$$w_o \leq M x_o \tag{8.22b}$$
$$w_o \geq z - M(1 - x_o) \tag{8.22c}$$
$$w_o \geq 0 \tag{8.22d}$$

Neste caso, M deve ser um majorante para z (i.e., $M \geq z_{max}$). Se $z_{min} = 0$, esta formulação (8.22) com $M = z_{max}$ é equivalente à formulação de McCormick mencionada acima (e, portanto, exata). No entanto, para o caso geral onde z pode ser negativo, a primeira formulação (baseada diretamente em McCormick) ou uma adaptação cuidadosa da segunda é necessária.

A escolha de M é crucial: deve ser grande o suficiente para não cortar soluções ótimas, mas não tão excessivamente grande a ponto de causar instabilidades numéricas ou uma relaxação linear fraca (Pedroso, 2024). A aplicação dessas técnicas de linearização pode aumentar o tamanho do modelo (e.g., adicionando restrições), o que é uma consideração prática em problemas de grande escala, como mencionado em estudos sobre robustez numérica de métodos de otimização fracionária que empregam tais linearizações (conforme discutido em contextos como o da "Variável Inversa" que pode requerer linearização de produtos $z \cdot x_o$).

Em problemas ainda mais complexos, onde os termos a serem multiplicados são eles próprios somatórios ou combinações não lineares, técnicas mais avançadas, incluindo as **desigualdades de McCormick** para produtos gerais entre duas variáveis contínuas, são indispensáveis para preservar a integridade do modelo (McCormick, 1976).

8.5.3 Desigualdades de McCormick

As desigualdades de McCormick, introduzidas por Garth P. McCormick em 1976 (McCormick, 1976), são uma técnica fundamental para a linearização de produtos não lineares entre variáveis (contínuas e/ou binárias) em problemas de otimização, especialmente em Programação Linear Inteira Mista (MILP). Elas permitem substituir termos multiplicativos por um conjunto de restrições lineares que definem o envoltório convexo[10] mais justo (tightest) possível para o produto, assegurando que a solução do modelo linearizado seja equivalente à do modelo original não linear, dentro dos limites especificados para as variáveis (Wolsey and Nemhauser, 1999, Sherali and Adams, 1999).

Considere o produto:

$$w = x \cdot z,$$

onde x é uma variável binária ($x \in \{0, 1\}$) e z é uma variável contínua limitada por $\underline{z} \leq z \leq \overline{z}$ (onde $\underline{z}$ e $\overline{z}$ são z_{min} e z_{max} respectivamente). As desigualdades de McCormick para este produto específico

[10] O *envoltório convexo* (ou *convex hull*, em inglês) de um conjunto de pontos ou do gráfico de uma função é definido como a menor região convexa que contém todos esses pontos ou o gráfico original (Wolsey and Nemhauser, 1999). No contexto da linearização de produtos bilineares $w = x \cdot y$, onde $x \in [x^L, x^U]$ e $y \in [y^L, y^U]$, as desigualdades de McCormick descrevem matematicamente as facetas do envoltório convexo do conjunto não convexo $\{(x, y, w) \mid w{=}xy, x^L \leq x \leq x^U, y^L \leq y \leq y^U\}$. Essa aproximação é a mais "justa" (tightest) possível usando restrições lineares e é crucial para construir relaxações lineares exatas no espaço das variáveis relaxadas, garantindo que nenhuma solução válida do modelo original seja descartada e que, na ausência de outras não-linearidades ou integralidade, a solução da relaxação linear corresponda à do problema original (McCormick, 1976, McCormick, Wolsey and Nemhauser, 1999). As quatro inequações lineares que definem este envoltório são:

$$w \geq x^L y + y^L x - x^L y^L$$
$$w \geq x^U y + y^U x - x^U y^U$$
$$w \leq x^L y + y^U x - x^L y^U$$
$$w \leq x^U y + y^L x - x^U y^L$$

Estas garantem a equivalência dentro do domínio retangular definido pelos limites de x e y.

são obtidas aplicando os limites de x ($x^L = 0, x^U = 1$) nas fórmulas gerais, resultando em:

$$w \leq \overline{z} \cdot x \tag{8.23a}$$
$$w \geq \underline{z} \cdot x \tag{8.23b}$$
$$w \leq z - \underline{z} \cdot (1 - x) \tag{8.23c}$$
$$w \geq z - \overline{z} \cdot (1 - x) \tag{8.23d}$$

Analisando o comportamento dessas restrições:

- Quando $x = 0$: As inequações (8.23a) e (8.23b) tornam-se $w \leq 0$ e $w \geq 0$ (se $\underline{z} \geq 0$) ou $w \leq 0$ e $w \geq \underline{z} \cdot 0 = 0$ (mais geralmente, $0 \cdot \overline{z} \geq w \geq 0 \cdot \underline{z}$, forçando $w = 0$ if $\underline{z}, \overline{z}$ são finitos). As inequações (8.23c) e (8.23d) tornam-se $w \leq z - \underline{z}$ e $w \geq z - \overline{z}$. Combinando, $w = 0$ é a única solução. (Da (8.23a) $w \leq 0$, da (8.23b) $w \geq 0$, se $\underline{z} \cdot x$ for interpretado como 0 quando $x = 0$ independentemente de $\underline{z}$ ser infinito. Mais rigorosamente, quando $x = 0$, as quatro restrições forçam $w = 0$).
- Quando $x = 1$: As inequações (8.23a) e (8.23b) tornam-se $w \leq \overline{z}$ e $w \geq \underline{z}$. As inequações (8.23c) e (8.23d) tornam-se $w \leq z$ e $w \geq z$. Combinando, $w = z$ é forçado, e z já deve estar entre $\underline{z}$ e $\overline{z}$ por definição.

Dessa forma, o produto é linearizado sem perda de exatidão, dentro dos limites estabelecidos para z, o que é crucial para a modelagem precisa e eficiente de problemas que combinam decisões binárias com variáveis contínuas.

No contexto do nosso problema (WOP) ou de problemas fracionários linearizados (e.g., pelos métodos da Variável Inversa ou Charnes-Cooper, onde variáveis como t ou z multiplicam outras, potencialmente binárias, após a transformação), essa abordagem é particularmente útil. Produtos como $t \cdot y_a$ ou $z \cdot x_o$ podem surgir e necessitar de tal linearização. Com isso, as desigualdades de McCormick permitem que esses produtos sejam tratados de forma linear, preservando a integridade do modelo e garantindo que as soluções obtidas sejam válidas e otimizadas. Seu uso é recomendado especialmente quando os limites das variáveis contínuas são bem definidos e finitos.

8.5.4 O Método 2: Charnes-Cooper

A transformação de Charnes-Cooper, desenvolvida por Abraham Charnes e William W. Cooper (Charnes and Cooper, 1962), é considerada, na literatura, uma das abordagens mais elegantes e matematicamente rigorosas para converter problemas de PLF em PL equivalentes (Maxwell, 2014, Duarte, 2019). Essa técnica não apenas solucionou o desafio clássico da área[11], mas também abriu caminho para toda uma classe de métodos de linearização de funções objetivo fracionárias (Charnes and Cooper, 1962, Boueri et al., 2019, Almeida and Rebelatto, 2019). No contexto do nosso estudo, ela merece destaque especial devido à sua efetividade e aplicabilidade direta. Por essa razão, apresentamos a seguir alguns detalhes da sua aplicação ao nosso caso de estudo.

O cerne da transformação de Charnes-Cooper está na introdução de uma variável contínua t que, diferentemente da variável z anterior, não serve apenas como o inverso do denominador da função

[11] O principal desafio clássico que a transformação de Charnes-Cooper resolveu está relacionado à natureza não linear dos problemas de Programação Linear Fracionária (PLF). Nestes problemas, a função objetivo é uma razão entre duas funções lineares, o que impede a aplicação direta dos métodos tradicionais de Programação Linear (PL), pois a presença da fração torna o problema não linear e, portanto, mais complexo de resolver.

A transformação proposta por Charnes e Cooper (Charnes and Cooper, 1962) consiste em uma mudança de variáveis que fixa o denominador da função objetivo igual a 1, convertendo o problema original em um problema de programação linear equivalente. Dessa forma, o problema fracionário passa a ser tratado com as mesmas técnicas rigorosas e eficientes da programação linear clássica, superando a dificuldade imposta pela não linearidade da função objetivo.

Esse avanço foi fundamental para diversas áreas, como a Análise Envoltória de Dados (DEA), onde o modelo CCR utiliza essa transformação para avaliar a eficiência relativa de unidades produtivas, linearizando a função objetivo que originalmente é uma razão entre outputs e inputs (Maxwell, 2014, Coelli et al., 1998, Charnes et al., 1978).

fracionária. Aqui a variável nova desempenha um papel maior, servindo de fator de escala que permite reescrever todo o problema em uma nova base de variáveis, de forma sistemática, homogênea e, sobretudo, linear (Marques Júnior, 2021).

$$t = \frac{1}{D(y)} = \frac{1}{1 + \sum_{a \in \mathcal{A}} y_a} \tag{8.24}$$

A partir disso, transformamos todas as variáveis do modelo original. Em termos práticos, significa multiplicá-las também por t, obtendo, por conseguinte, um conjunto de variáveis escaladas:

$$x'_o = t \cdot x_o, \quad \forall o \in \mathcal{O} \tag{8.25}$$
$$y'_a = t \cdot y_a, \quad \forall a \in \mathcal{A} \tag{8.26}$$

Essa transformação tem uma vantagem prática fundamental: é válida sempre que o denominador da função fracionária original for estritamente positivo. No nosso caso, isso está garantido graças à inclusão daquela constante "+1" robustez[12] no denominador da função objetivo, que assegura que $D(y) > 0$ para qualquer combinação de $y_a \in \{0, 1\}$, evitando o problemas como a divisão por zero ou indefinições que, em modelos reais, podem ser fatais para o solver (Schaible, 1981).

Aplicando essa transformação ao nosso problema, conforme descrito originalmente em (Charnes and Cooper, 1962, Maxwell, 2014), cuja função objetivo era:

$$\max Z = \frac{\sum_{o \in \mathcal{O}} S_o \cdot x_o}{1 + \sum_{a \in \mathcal{A}} y_a} \tag{8.27}$$

sujeita às restrições estabelecidas no Desafio[13], obtemos a seguinte versão transformada e linearizada:

$$\max Z' = \sum_{o \in \mathcal{O}} S_o \cdot x'_o \tag{8.28}$$

Naturalmente, mas não tão óbvio, essa nova formulação exige um novo conjunto de restrições. Todas as restrições originais do Desafio são multiplicadas por t e reescritas no novo sistema de variáveis. As principais incluem:

- A chamada restrição de normalização, que garante que $t \cdot (1 + \sum y_a) = 1$ e, por conseguinte, a equivalência entre os modelos:
$$t + \sum_{a \in \mathcal{A}} y'_a = 1 \tag{8.29}$$

- Restrições que limitam as variáveis escaladas em função de t, auxiliando na preservação da natureza das variáveis originais após a recuperação:
$$x'_o \leq t, \quad \forall o \in \mathcal{O} \tag{8.30}$$
$$y'_a \leq t, \quad \forall a \in \mathcal{A} \tag{8.31}$$

- A transformação das restrições do problema original. A **restrição de limite inferior de unidades**, por exemplo, que originalmente é:
$$\sum_{o \in \mathcal{O}} S_o \cdot x_o \geq LB \tag{8.32}$$
transforma-se em:
$$\sum_{o \in \mathcal{O}} S_o \cdot x'_o \geq LB \cdot t \tag{8.33}$$

[12] Vide nota 3 do capítulo original, referente à robustez adicionada pela constante +1.
[13] Vide subcapítulo Restrições do Modelo do capítulo original.

- Similarmente, se houver uma **restrição de limite superior de unidades** no Desafio, como $\sum_{o\in\mathcal{O}} S_o \cdot x_o \leq UB$, sua forma transformada seria (considerando um possível termo de folga $\delta_1 \geq 0$, se aplicável):

$$\sum_{o\in\mathcal{O}} S_o \cdot x'_o \leq UB \cdot t + \delta_1 \tag{8.34}$$

- Outras restrições, como uma de **cobertura de estoque** (assumindo que $I(o)$ são os itens no pedido o, $I(a)$ os itens acessíveis pelo corredor a, e AV_{ai} a disponibilidade do item i no corredor a, conforme definido no Desafio), do tipo $\sum_{o\,:\,i\in I(o)} x_o \leq \sum_{a\,:\,i\in I(a)} AV_{ai}\, y_a$, seria transformada em:

$$\sum_{o\,:\,i\in I(o)} x'_o \leq \sum_{a\,:\,i\in I(a)} AV_{ai}\, y'_a, \quad \forall\, i \in I \tag{8.35}$$

Adicionalmente, as variáveis transformadas devem respeitar os seguintes domínios[14]:

$$x'_o \geq 0, \quad \forall o \in \mathcal{O} \tag{8.36}$$
$$y'_a \geq 0, \quad \forall a \in \mathcal{A} \tag{8.37}$$
$$t > 0 \tag{8.38}$$

A restrição $t > 0$ é essencial, pois, sem ela o fator de escala se tornaria inválido e a transformação perderia sentido. As condições $x'_o \leq t$ e $y'_a \leq t$ (de (8.30) e (8.31)), combinadas com a não-negatividade $x'_o \geq 0$ e $y'_a \geq 0$, implicam que $x'_o \in [0, t]$ e $y'_a \in [0, t]$. Para que as variáveis originais x_o e y_a (recuperadas por $x_o = x'_o/t$ e $y_a = y'_a/t$) assumam valores binários (0 ou 1), é crucial que x'_o e y'_a resultem em 0 ou t na solução ótima. Esse detalhe é importante porque preserva o caráter inteiro e binário do modelo original, o que é especialmente relevante em problemas de Programação Inteira Mista com função objetivo fracionária (Maxwell, 2014).

Esse comportamento desejado das variáveis transformadas ($x'_o, y'_a \in \{0, t\}$), no entanto, exige um cuidado adicional: embora as restrições impostas ajudem, nada garante, por si só, que x'_o e y'_a serão automaticamente 0 ou t pelo solver. Por isso, pode ser necessário impor condições extras no modelo — um ponto que nos leva ao tratamento das variáveis binárias após a transformação. A literatura apresenta duas abordagens clássicas para lidar com esse tipo de estrutura após a transformação:

- **Restrições adicionais explícitas:** Além de $x'_o \leq t$ e $x'_o \geq 0$, pode-se usar uma variável binária auxiliar z_o (idealmente, z_o seria a própria x_o original, se o solver permitir essa construção mista

[14] Em modelos de otimização PL, PLI ou MILP, as variáveis de decisão sempre estão associadas a um **domínio**, ou seja, ao conjunto de valores possíveis que cada variável pode assumir. As *restrições de domínio das variáveis* consistem na explicitação, no modelo matemático, desses conjuntos de valores, limitando a região factível apenas aos pontos cujas variáveis satisfaçam tais condições.

Por exemplo:

- determinada variável deve ser não-negativa e inteira:

$$x_i \geq 0, \quad x_i \in \mathbb{Z}$$

- outra pode ser binária:

$$y_j \in \{0, 1\}$$

- e até as de proporção podem ser restritas a um intervalo unitário:

$$0 \leq p_k \leq 1$$

Essas restrições são normalmente explicitadas após as equações principais do modelo, como fizemos várias vezes aqui, geralmente ao final da formulação, indicando, ou tentando indicar, os limites impostos a cada uma ou a a um grupo de variáveis de Souza et al. (2018). Logo, ao resolver um problema de otimização, os algoritmos levam em consideração não apenas as restrições relacionais do problema (igualdades e desigualdades), mas também os domínios declarados para cada variável. Essas últimas, aliás, são essenciais, pois impedem que soluções não factíveis — tais como quantidades negativas, frações de elementos que só podem ser inteiros ou decisões não-binárias — sejam consideradas candidatas à solução ótima. Dito como aprendemos em aula, essas restrições de domínio definem o espaço viável do problema, restringindo o conjunto de soluções admissíveis (Kolman, 1995, de Souza et al., 2018).

ou se for uma etapa de pré-processamento):

$$
\begin{aligned}
& x'_o \leq t, \quad \forall o \in \mathcal{O} \\
& x'_o \geq t - M(1 - z_o), \quad \forall o \in \mathcal{O} \\
& x'_o \leq M z_o, \quad \forall o \in \mathcal{O} \qquad\qquad (\text{para garantir } x'_o = 0 \text{ se } z_o = 0)
\end{aligned}
$$

onde z_o é uma variável binária e M é uma constante suficientemente grande (idealmente, um majorante para t). Similarmente para y'_a.

- **Substituição direta (com linearização subsequente):** Introduzir variáveis binárias z_o e w_a e definir:

$$
\begin{aligned}
x'_o &= z_o \cdot t \\
y'_a &= w_a \cdot t
\end{aligned}
$$

Essa técnica é mais intuitiva, mas reintroduz produtos bilineares ($t \cdot z_o$, $t \cdot w_a$), o que nos obriga a usar outras técnicas de linearização (como as de McCormick ou Glover) para manter o modelo linear.

Segundo Schaible e Ibaraki (Schaible and Ibaraki, 1983), a primeira abordagem (com restrições adicionais bem formuladas) tende a ser mais estável do ponto de vista numérico e oferece melhor desempenho prático, especialmente quando implementada em solvers comerciais como CPLEX ou Gurobi (IBM, 2017, Gurobi Optimization, LLC, 2023).

Esta transformação resulta na adição de $1+|\mathcal{O}|+|\mathcal{A}|$ variáveis contínuas (t, x'_o, y'_a), uma restrição de igualdade para normalização (8.29), e vínculos lineares como (8.30) e (8.31) que, junto com as demais restrições transformadas, buscam preservar a binariedade das variáveis originais após a recuperação.

Por fim, uma das maiores vantagens da transformação de Charnes-Cooper é a simplicidade da reconstrução da solução original:

$$x_o = \frac{x'_o}{t}, \quad \forall o \in \mathcal{O} \tag{8.39}$$

$$y_a = \frac{y'_a}{t}, \quad \forall a \in \mathcal{A} \tag{8.40}$$

Como o objetivo é que $x'_o, y'_a \in \{0, t\}$ na solução ótima e $t > 0$, essa recuperação sempre retorna valores binários (0 ou 1), preservando a coerência com o modelo original.

Veja o algoritmo abaixo, que resume o processo de transformação acima aplicado ao nosso problema. É uma versão simplificada do original, mas mantém a essência da técnica, podendo ser facilmente adaptado para diferentes contextos e problemas fracionários, desde que as condições de normalização e linearidade sejam respeitadas.

Algoritmo 2: Transformação de Charnes-Cooper adaptada ao WOP

Conjuntos $\mathcal{O}$, $\mathcal{A}$, unidades S_o, parâmetros de restrição (LB, UB, AV_{ai} etc.), tolerância ε
Solução ótima em variáveis escaladas (x'_o, y'_a, t)

Defina $t > 0$ e as variáveis escaladas:;
$x'_o = t \cdot x_o, \quad y'_a = t \cdot y_a, \quad \forall o \in \mathcal{O},\ a \in \mathcal{A}$;

Imponha a restrição de normalização:;
$t + \sum_{a \in \mathcal{A}} y'_a = 1$;

Reescreva as restrições originais, multiplicadas por t (exemplos):; `// Vide eqs.` (8.33)-(8.35)
$\sum_{o \in \mathcal{O}} S_o \cdot x'_o \geq LB \cdot t$;
$\sum_{o \in \mathcal{O}} S_o \cdot x'_o \leq UB \cdot t + \delta_1$;
$\sum_{o\,:\,i \in I(o)} x'_o \leq \sum_{a\,:\,i \in I(a)} AV_{ai}\, y'_a$;
... (demais restrições do Desafio transformadas);

Restrinja os domínios das variáveis escaladas:; `// Vide eqs.` (8.30)-(8.38)
$0 \leq x'_o \leq t, \quad \forall o \in \mathcal{O}$;
$0 \leq y'_a \leq t, \quad \forall a \in \mathcal{A}$;
$t > \varepsilon$ (para evitar divisão por zero na prática, ε pequeno);

Pode ser necessário adicionar restrições para garantir $x'_o, y'_a \in \{0, t\}$ (ver discussão no texto).

Maximize:;
$\sum_{o \in \mathcal{O}} S_o \cdot x'_o$;

Vamos tentar explicar essa transformação de forma mais intuitiva usando uma analogia com o exemplo do armazém do Mercado Livre que mencionamos no início. Depois de definida a nova variável contínua t, que representa o esforço ou o "custo embutido" no denominador do problema fracionário, o algoritmo executa um reescalonamento sistemático de todas as variáveis do modelo. Isso significa que, ao invés de atuar diretamente sobre a razão, transformamos o problema em um novo ambiente, onde as variáveis escaladas já incorporam a fração original — permitindo que o modelo seja resolvido com ferramentas lineares clássicas.

Cada etapa do algoritmo tem uma interpretação intuitiva:

- Ao definir $x'_o = t \cdot x_o$ e $y'_a = t \cdot y_a$, estamos comprimindo (ou expandindo) as decisões em função do nível de esforço representado por t.
- A condição $t + \sum_{a \in \mathcal{A}} y'_a = 1$ garante que a escala escolhida seja coerente com a realidade, já que impõe uma régua única sobre todas as variáveis.
- As restrições originais são mantidas, mas agora adaptadas à nova escala. Por exemplo, a soma, para todo pedido o no conjunto $\mathcal{O}$, do número de unidades S_o multiplicado pela variável escalada x'_o, deve ser maior ou igual a LB vezes o fator de escala t

$$\sum_{o \in \mathcal{O}} S_o \cdot x'_o \geq LB \cdot t \tag{8.41}$$

  quer dizer que ainda é necessário coletar um número mínimo de unidades — mas esse mínimo agora é proporcional ao esforço total permitido.
- A função objetivo, ao fim e ao cabo, resta linear:

$$\max \sum_{o \in \mathcal{O}} S_o \cdot x'_o \tag{8.42}$$

Uma vez obtida a solução ótima do problema escalado, basta reverter o reescalonamento para recuperar as decisões originais:

$$x_o = \frac{x'_o}{t}, \quad \forall o \in \mathcal{O} \tag{8.43}$$

$$y_a = \frac{y'_a}{t}, \quad \forall a \in \mathcal{A} \tag{8.44}$$

Considere:

$$\max \frac{2x}{1+y}, \quad x \in \{0,1\},\ y \in \{0,1\} \tag{8.45}$$

Defina $t = \frac{1}{1+y}$, e aplique a transformação:

$$x' = t \cdot x, \quad y' = t \cdot y, \quad t + y' = 1 \tag{8.46}$$

O objetivo se torna:

$$\max 2x', \quad \text{com } x' \in [0,t],\ y' \in [0,t],\ t > 0 \tag{8.47}$$

(e, idealmente, x', y' resultando em 0 ou t na solução ótima para que x, y sejam binárias).

A recuperação da solução original é imediata:

$$x = \frac{x'}{t}, \quad y = \frac{y'}{t} \tag{8.48}$$

Ainda que, à primeira vista, essa transformação possa parecer sofisticada, para não dizer difícil, ela é reconhecida na literatura como uma técnica simples, como demonstrado nos trabalhos originais dos autores (Charnes and Cooper, 1962) e nos estudos posteriores (Ferreira, 2015). Entretanto, é importante esclarecer um ponto técnico que costuma ser mal interpretado: embora a reformulação pareça garantir que as variáveis recuperadas serão automaticamente binárias (isto é, 0 ou 1), isso nem sempre ocorre de forma natural. Para que a integridade binária das variáveis seja preservada, é necessário impor restrições adicionais ao modelo transformado — como as discutidas para assegurar que $x'_o, y'_a \in \{0, t\}$ na solução ótima e que o denominador seja estritamente positivo, assegurando a viabilidade da inversão e da reinterpretação das variáveis escaladas[15].

Mas talvez isso tudo ainda soe meio abstrato. O exemplo prometido pode ajudar. Imagine-se novamente dentro do armazém do Mercado Livre, tentando decidir quais pedidos vale a pena coletar e por quais corredores passar. Só que, desta vez, em vez de aplicar um redutor diretamente sobre os pedidos, como fizeste antes, você resolve redefinir a escala do problema inteiro. Você inteligentemente pensa: "E se eu medir tudo — pedidos, corredores, restrições — com base no *quanto estou disposto a me esforçar*?" Como já dissemos, você é um gênio. Esse esforço total é representado por uma variável contínua t, definida como:

$$t = \frac{1}{1 + \sum_{a \in \mathcal{A}} y_a} \tag{8.49}$$

Logo, quanto mais corredores forem visitados, menor será o valor de t, e todo o modelo será proporcionalmente reescalado — como se o esforço envolvido comprimisse as decisões possíveis. O método de Charnes-Cooper, portanto, não busca balancear numerador e denominador diretamente. Ele muda o

[15] A literatura recomenda, nesse contexto, o uso de restrições de acoplamento ou técnicas como as desigualdades de McCormick para assegurar que a transformação mantenha as variáveis no conjunto binário após a divisão pelo fator de escala t. Vide (Ferreira, 2015).

cenário onde o problema é resolvido: transforma uma razão complicada em uma simples soma linear. É como se você dissesse: "não vou lidar com a fração — vou mudar o sistema de medidas" e, com isso, você ainda ganha diversas vantagens práticas, como a estabilidade numérica e a compatibilidade com variáveis inteiras, a facilidade de recuperação da solução original e, principalmente, uma linearização exata — sem necessidade de métodos iterativos ou aproximações não lineares.

Se você recordar bem do exemplo anterior sobre a variável inversa, vai perceber que os dois métodos compartilham uma filosofia de linearização adaptativa semelhante. A diferença, no entanto, está na forma que cada um propõe. Aquele impõe um redutor diretamente sobre os pedidos — cada x_o é multiplicado por uma variável contínua z, que representa o inverso do esforço (ou seja, do número de corredores envolvidos). Esse atua de forma mais ampla, já que, em vez de aplicar um redutor localizado, redefine o sistema de medidas de todo o problema, reconstruindo o universo decisório a partir do fator de escala global t, proporcional ao esforço total estimado, de tal modo que todas as variáveis sejam ajustadas de acordo com essa nova régua.

Grosso modo, *Enquanto a variável inversa aplica uma lente sobre os pedidos, Charnes-Cooper muda o tamanho do universo onde a decisão é tomada.* Para destacar as diferenças entre essas abordagens de forma mais objetiva, apresentamos a seguir uma comparação entre os métodos discutidos: a variável inversa, a transformação de Charnes-Cooper e, na sequência, o algoritmo iterativo de Dinkelbach.

Tabela 2: Comparação entre técnicas de linearização para problemas fracionários

Critério	**Variável Inversa**	**Charnes-Cooper**	**Dinkelbach**
Transformação	Parcial (denominador)	Completa (todas as variáveis)	Iterativa
Estabilidade Numérica	Moderada	Alta	Dependente da convergência
Variáveis Adicionais	$O(\|\mathcal{O}\|)$	$1 + \|\mathcal{O}\| + \|\mathcal{A}\|$ contínuas	Nenhuma (por iteração)
Compatibilidade com PLI	Requer adaptações	Direta com adaptações (para garantir binariedade)	Preserva estrutura inteira
Eficiência Computacional	Moderada	Alta (problemas pequenos/médios)	Alta (problemas grandes)

A transformação da variável inversa, como vimos antes, atua no denominador da função objetivo e gera estabilidade numérica moderada para problemas de PLI. A de Charnes-Cooper, por sua vez, reformula completamente o problema, escalando todas as variáveis pelo fator contínuo adicionado, proporcionando estabilidade numérica mais elevada e compatibilidade direta com PLI (com as devidas restrições para assegurar a binariedade das variáveis recuperadas), apesar de aumentar o número de variáveis. Talvez por isso seja mais adequada para problemas pequenos e médios (Charnes and Cooper, 1962). O Dinkelbach, que veremos a seguir, é iterativo, mas não adiciona variáveis extras por iteração, como o nome parece sugerir[16], porquanto preserva a estrutura inteira do problema, sendo eficiente especialmente para problemas de grande porte, apesar de sua estabilidade depender da convergência do algoritmo (Dinkelbach, 1967), como explicaremos no próximo capítulo. Essa comparação, ainda que simples, contribui, do ponto de vista prático, para a escolha da técnica mais adequada, de acordo com o tamanho do problema, da estrutura das variáveis e da necessidade de estabilidade numérica ou não.

[16] O método de Dinkelbach é iterativo porque resolve uma sequência de problemas auxiliares, ajustando um parâmetro a cada iteração até atingir a convergência. No entanto, em cada iteração, o problema resolvido mantém exatamente as mesmas variáveis e restrições do problema original, apenas modificando a função objetivo. Assim, o método não adiciona variáveis extras por iteração, o que contribui para sua eficiência e simplicidade estrutural, especialmente em problemas de grande porte. Por outro lado, sua estabilidade depende da convergência do algoritmo, que pode variar conforme o problema (Dinkelbach, 1967).

8.5.5 O Algoritmo Iterativo de Dinkelbach

Por fim, chegamos ao Dinkelbach — uma ferramenta frequentemente descrita na literatura como "poderosa" para a resolução de problemas de programação fracionária. O motivo? Ele transforma a complexidade da razão em uma sequência de problemas paramétricos mais simples, todos linearizados (ou linearizáveis), todos tratáveis por solvers modernos. Sua principal virtude, segundo o autor do método, está na capacidade de convergir rapidamente para a solução ótima do problema original, mesmo em casos de grande porte — como algumas das instâncias que enfrentamos neste trabalho.

Partindo do nosso problema fracionário:

$$\max Z = \frac{N(\mathbf{x}, \mathbf{y})}{D(\mathbf{x}, \mathbf{y})}, \tag{8.50}$$

onde $N(\mathbf{x}, \mathbf{y})$ representa o numerador (e.g., a quantidade total de unidades coletadas, $N(\mathbf{x}) = \sum_{o \in \mathcal{O}} S_o x_o$) e $D(\mathbf{x}, \mathbf{y})$ o denominador (e.g., o total de corredores visitados mais um, $D(\mathbf{y}) = 1 + \sum_{a \in \mathcal{A}} y_a$), a ideia central do método de Dinkelbach é transformar o problema fracionário em uma sequência de problemas paramétricos, mais simples de resolver Dinkelbach (1967), You and Grossmann (2009). Em vez de maximizar a razão diretamente, o método propõe resolver iterativamente um problema auxiliar $P(\lambda)$, onde λ (também denotado como ρ em alguma literatura) é um parâmetro que representa uma estimativa do valor ótimo da função objetivo fracionária. Este parâmetro é ajustado a cada passo do algoritmo, até que a função objetivo auxiliar do problema paramétrico atinja um valor suficientemente próximo de zero (indicando convergência), momento em que a solução encontrada é considerada ótima para o problema original Dinkelbach (1967).

Formalmente, o problema auxiliar em cada iteração k, para um dado parâmetro λ_k, é dado por:

$$P(\lambda_k): \quad \max \quad f(\lambda_k) = N(\mathbf{x}, \mathbf{y}) - \lambda_k D(\mathbf{x}, \mathbf{y}), \tag{8.51}$$

sujeito às restrições originais do problema. Se forem utilizadas penalidades suaves para flexibilizar certas restrições (e.g., $\delta_1, \delta_2 \geq 0$ como folgas para limites de unidades ou volume, com custos $\pi_1, \pi_2 > 0$), a função objetivo do subproblema torna-se:

$$P(\lambda_k): \quad \max \quad f(\lambda_k) = N(\mathbf{x}, \mathbf{y}) - \lambda_k D(\mathbf{x}, \mathbf{y}) - \pi_1 \delta_1 - \pi_2 \delta_2. \tag{8.52}$$

A condição de parada clássica do algoritmo é quando $f(\lambda_k)$ está suficientemente próximo de zero. Se (x^*, y^*) é a solução ótima do problema paramétrico para $\lambda_k = \lambda^*$, então

$$N(x^*, y^*) - \lambda^* D(x^*, y^*) \approx 0, \tag{8.53}$$

o que implica que

$$\lambda^* \approx \frac{N(x^*, y^*)}{D(x^*, y^*)}, \tag{8.54}$$

isto é, λ^* é (aproximadamente) o valor ótimo da função objetivo fracionária original.

Para o nosso problema específico (WOP), a função objetivo paramétrica em cada iteração, incorporando as penalidades suaves, assume a seguinte forma:

$$\max \quad \sum_{o \in \mathcal{O}} S_o \cdot x_o - \lambda_k \left(1 + \sum_{a \in \mathcal{A}} y_a\right) - \pi_1 \delta_1 - \pi_2 \delta_2, \tag{8.55}$$

sujeita a todas as restrições do modelo, como os limites de unidades na *wave* (possivelmente flexibilizados por δ_1, δ_2), a cobertura de itens, entre outras condições previamente definidas. É importante observar que, em cada iteração, o problema $P(\lambda_k)$ permanece um problema de programação linear inteira mista (MILP), o que permite sua resolução por *solvers* comerciais como o CPLEX — utilizado

neste estudo. Outra vantagem é que ele não requer a introdução de variáveis contínuas extras para a linearização da fração a cada iteração (exceto as possíveis variáveis de folga δ_1, δ_2, que são parte do modelo base), o que preserva a estrutura do problema original e o torna particularmente eficiente em instâncias de grande porte. A avaliação dos termos $N(\mathbf{x})$ e $D(\mathbf{y})$ pode, inclusive, ser acelerada em hardware específico como GPUs (e.g., usando CuPy), reduzindo drasticamente o tempo de cada iteração em problemas muito grandes.

No entanto, vale destacar que a estabilidade numérica e a velocidade de convergência, embora geralmente boas, podem ser influenciadas pela formulação e pela natureza do problema Dinkelbach (1967), Duarte (2019). O método de Dinkelbach é provadamente convergente e, na prática, exibe convergência rápida, frequentemente superlinear:[17]

"$q_{k+1} > q_k \ \ldots \ \lim_{k\to\infty} q_k = q^*$" (adaptado de Dinkelbach (1967))

Quando o método converge adequadamente, os resultados são estáveis e confiáveis.

Para facilitar a compreensão, apresentamos abaixo o pseudocódigo clássico do Dinkelbach, adaptado para a resolução do nosso problema, incluindo a gestão de penalidades suaves. As adaptações consistem principalmente na especificação das funções $N(\mathbf{x})$ e $D(\mathbf{y})$, e na inclusão das restrições do problema (com folgas) que devem ser consideradas em cada iteração do solver de MILP. O núcleo do algoritmo permanece o mesmo: **resolver iterativamente um problema linear (ou MILP) paramétrico e ajustar o parâmetro λ até a convergência.**

Algoritmo 3: Algoritmo de Dinkelbach para o WOP (com penalidades suaves)

Conjunto de pedidos $\mathcal{O}$, corredores $\mathcal{A}$, unidades por pedido S_o, unidades de item por pedido U_{oi}, disponibilidade por corredor AV_{ai}, limites LB, UB, custos de penalidade π_1, π_2, tolerância ε, máximo de iterações MaxIter Valor ótimo λ^*, solução ótima (x^*, y^*), folgas (δ_1^*, δ_2^*)

Inicialize $\lambda_0 \leftarrow 0$ (ou uma estimativa heurística, e.g., baseada em $S_o/|R_o|$), $k \leftarrow 0$;

repeat

  $k \leftarrow k + 1$;

  Resolva o problema paramétrico $P(\lambda_k)$:;

    $\max \sum_{o\in\mathcal{O}} S_o x_o - \lambda_k \left(1 + \sum_{a\in\mathcal{A}} y_a\right) - \pi_1\delta_1 - \pi_2\delta_2$;

  sujeito a:;

    $\sum_{o\in\mathcal{O}} S_o x_o \geq LB - \delta_2$;

    ; // Exemplo: δ_2 para folga em LB

    $\sum_{o\in\mathcal{O}} S_o x_o \leq UB + \delta_1$;

    ; // Exemplo: δ_1 para folga em UB

    $\sum_{o\in\mathcal{O}} U_{oi} x_o \leq \sum_{a\in\mathcal{A}} AV_{ai} y_a, \ \forall i \in \mathcal{I}$;

    $x_o \in \{0,1\}, \ \forall o \in \mathcal{O}$;

    $y_a \in \{0,1\}, \ \forall a \in \mathcal{A}$;

    $\delta_1 \geq 0, \delta_2 \geq 0$;

    (Outras restrições do WOP);

  Obtenha a solução ótima $(x_k^*, y_k^*, \delta_{1,k}^*, \delta_{2,k}^*)$ e o valor ótimo $f(\lambda_k)$ do subproblema;

  Calcule:;

    $N_k^* = \sum_{o\in\mathcal{O}} S_o x_{o,k}^*$;

    $D_k^* = 1 + \sum_{a\in\mathcal{A}} y_{a,k}^*$;

  if $D_k^* \leq$

Vamos ao passo a passo:

[17] Conforme Dinkelbach (1967), o parâmetro λ_k (ou q_k na notação original) avança monotonicamente (ou de forma bem comportada) em direção ao valor ótimo λ^* (ou q^*), com $\lim_{k\to\infty} \lambda_k = \lambda^*$. A literatura reporta que, para muitas aplicações, poucas iterações são necessárias para atingir uma boa precisão (e.g., 8–10 iterações para $\varepsilon \leq 10^{-4}$ em alguns casos).

1. **Inicialização:** começamos com um valor inicial para o parâmetro λ, usualmente $\lambda_0 = 0$. Alternativamente, pode-se usar uma estimativa heurística, por exemplo, ordenando os pedidos por uma razão de eficiência (como S_o dividido pelo número de corredores únicos que o pedido o requer, $|R_o|$) e construindo uma solução inicial para derivar λ_0. Esse valor funciona como um palpite inicial da razão ótima a ser alcançada.

2. **Construção do Problema Paramétrico:** a cada iteração k, o algoritmo constrói e resolve um problema auxiliar $P(\lambda_k)$, substituindo o objetivo fracionário por uma função linear (ou linear inteira mista) dependente de λ_k:

$$f(\lambda_k) = N(\mathbf{x}_k, \mathbf{y}_k) - \lambda_k D(\mathbf{x}_k, \mathbf{y}_k)(-\text{penalidades, se houver}), \tag{8.56}$$

   onde $N(\mathbf{x}_k, \mathbf{y}_k)$ e $D(\mathbf{x}_k, \mathbf{y}_k)$ representam, respectivamente, o numerador e o denominador da função objetivo original, avaliados na solução da iteração $k-1$ (para calcular λ_k) ou como variáveis na otimização de $P(\lambda_k)$. As variáveis $(\mathbf{x}_k, \mathbf{y}_k)$ indicam a solução corrente do subproblema.

3. **Resolução do Problema Auxiliar:** resolvemos o problema paramétrico $P(\lambda_k)$, ou seja, encontramos o vetor de decisão $(\mathbf{x}_k^*, \mathbf{y}_k^*)$ (e folgas $\delta_{1,k}^*, \delta_{2,k}^*$) que maximiza a expressão (8.55), respeitando todas as restrições do modelo original.

4. **Atualização do Parâmetro:** a partir da solução ótima $(\mathbf{x}_k^*, \mathbf{y}_k^*)$ obtida, calculamos um novo valor para λ:

$$\lambda_{k+1} = \frac{N(\mathbf{x}_k^*, \mathbf{y}_k^*)}{D(\mathbf{x}_k^*, \mathbf{y}_k^*)}. \tag{8.57}$$

   Esse valor representa uma nova estimativa da razão ideal, baseada na melhor solução encontrada na iteração corrente. É crucial garantir que $D(\mathbf{x}_k^*, \mathbf{y}_k^*) > 0$ (ou $D(\mathbf{x}_k^*, \mathbf{y}_k^*) >$

4. **Critério de Parada:** o processo se repete até que o valor da função objetivo do subproblema, $f(\lambda_k)$, esteja suficientemente próximo de zero — isto é,

$$|f(\lambda_k)| = |N(\mathbf{x}_k^*, \mathbf{y}_k^*) - \lambda_k D(\mathbf{x}_k^*, \mathbf{y}_k^*)(-\text{penalidades})| < \varepsilon, \tag{8.58}$$

   para uma tolerância pequena $\varepsilon > 0$, previamente definida. Esse valor controla a precisão desejada para a solução final. Quando essa condição é satisfeita (ou um número máximo de iterações é atingido), entendemos que a razão atingiu o valor ótimo buscado:

$$\lambda_k \approx \frac{N(\mathbf{x}_k^*, \mathbf{y}_k^*)}{D(\mathbf{x}_k^*, \mathbf{y}_k^*)}. \tag{8.59}$$

   O método, então, é interrompido, e $(\mathbf{x}_k^*, \mathbf{y}_k^*)$ é considerado uma solução ótima (ou suficientemente boa) para o problema fracionário original.

Isso demonstra que o algoritmo de Dinkelbach segue uma lógica simples, como defende a literatura, mas para entendê-la melhor, como cada passo se traduz na prática — e especialmente por que essa abordagem funciona tão bem —, vale revisitar o nosso já conhecido exemplo do armazém, refazendo os passos do algoritmo nele, dando forma concreta aos conceitos abstratos apresentados.

Imagine, mais uma vez, que você precisa coletar mercadorias no grande armazém do Mercado Livre e está com pressa, cansado de andar, afinal, esta já é a terceira vez que te pedimos isso. Seu objetivo, portanto, é ser mais eficiente ainda, ou seja, pegar a maior quantidade possível de itens, passando pelo menor número de corredores possível.

Apesar disso, pode ser que na primeira tentativa, você simplesmente tente pegar o máximo de itens possível, ignorando completamente a relação entre esforço e resultado. Mas daí você lembra do Dinkelbach e decide aplicá-lo para otimizar sua coleta.

Você começa com um palpite inicial para λ (digamos, $\lambda_0 = 0$): todo item coletado vale tanto quanto o custo de atravessar um corredor (ou, com $\lambda_0 = 0$, o custo dos corredores é inicialmente ignorado na

função objetivo paramétrica). Então, em vez de tentar maximizar a razão diretamente, você resolve um problema auxiliar em que o objetivo é maximizar a soma das unidades coletadas menos λ_0 vezes o "custo" dos corredores visitados (mais um) e menos quaisquer penalidades por violações suaves de restrições.

Após resolver esse problema, você obtém uma solução inicial com um conjunto de pedidos selecionados e os corredores correspondentes visitados. Com base nisso, você calcula o valor do numerador (quantidade total de unidades coletadas, N_0^*) e o valor do denominador (número de corredores visitados mais um, D_0^*). Com esses dois valores em mãos, você atualiza o parâmetro lambda: $\lambda_1 = N_0^*/D_0^*$.

Digamos que você coletou 10 itens e percorreu 2 corredores (logo $D_0^* = 1 + 2 = 3$), então o novo valor de lambda será $\lambda_1 = 10/3 \approx 3,33$. Isso significa que, na próxima rodada (iteração $k = 1$), cada "unidade de denominador" (corredor visitado + 1) terá um "custo" de 3,33 na função objetivo paramétrica, ponderando o numerador.

Com esse novo valor λ_1, você repete o processo: resolve novamente o problema auxiliar, agora usando λ_1 como multiplicador do denominador na função objetivo. O objetivo continua sendo maximizar o total ajustado da função, levando em conta essa nova ponderação.

Esse processo se repete iterativamente: resolve-se o problema com o λ_k atual, calcula-se uma nova razão $\lambda_{k+1} = N_k^*/D_k^*$ com a solução encontrada. O ciclo continua até que a diferença $N_k^* - \lambda_k D_k^*$ (mais penalidades) fique suficientemente próxima de zero — ou seja, até que o valor da função $f(\lambda_k)$ do subproblema indique convergência. Quando isso ocorre, significa que a razão ótima foi atingida, e a solução $(\mathbf{x}_k^*, \mathbf{y}_k^*)$ encontrada é considerada ótima para o problema fracionário original (Dinkelbach, 1967).

Ao final desse processo, você — cansado, mas satisfeito — percebe que percorreu o mínimo necessário do armazém, colheu a melhor quantidade possível de pedidos e fez tudo isso com uma eficiência que nem mesmo os algoritmos de roteamento do Mercado Livre conseguiriam bater. E o melhor: sem precisar testar todas as combinações possíveis ou ajustar manualmente pesos e penalidades (além dos π_j das folgas, se usadas).

Essa jornada iterativa, que começou com um palpite e evoluiu com ajustes progressivos da razão entre esforço e recompensa, é exatamente o que o algoritmo de Dinkelbach propõe — e o que o torna uma das abordagens mais interessantes e poderosas para resolver o nosso problema. Ufa! Agora que compreendemos os principais métodos para lidar com funções objetivo fracionárias, podemos apresentar a escolha metodológica que orientou nossa modelagem final para o Desafio SBPO 2025.

Mais do que uma decisão técnica, a seleção dos métodos adotados levou em conta fatores práticos como estabilidade numérica, custo computacional, escalabilidade e compatibilidade com os solvers disponíveis (em especial, o IBM CPLEX). Além disso, exploramos como a combinação estratégica entre esses métodos e o uso de aceleração via GPU (CUDA) pode oferecer vantagens concretas em instâncias de grande porte do Wave Order Picking (WOP). Vamos, portanto, ao que realmente interessa: como isso tudo se traduziu, na prática, na escolha e na implementação que fizemos.

9 Metodologia da Solução Proposta

A arquitetura de solução proposta para o Problema de Otimização de Coleta em Ondas (WOP) é fundamentada em um sistema modular. Esta abordagem integra sinergicamente a biblioteca **PuLP** para a formulação flexível de modelos matemáticos, o solver **CPLEX** para a resolução eficiente de Programas Lineares Inteiros Mistos (MILP), e a biblioteca **CuPy** para a aceleração de cômputos intensivos via Unidades de Processamento Gráfico (GPUs), especialmente na fase de pré-processamento. A metodologia se desdobra em módulos distintos e interconectados: leitura de instâncias, pré-processamento de dados (com variantes para CPU e GPU), construção do modelo matemático, execução da solução (incluindo o algoritmo de Dinkelbach para objetivos fracionários) e validação dos resultados.

A escolha desta pilha tecnológica reflete uma estratégia para balancear a agilidade no desenvolvi-

mento e prototipagem, proporcionada pelo ambiente Python e PuLP, com a capacidade de processamento de alto desempenho oferecida pelo CPLEX e pela aceleração em GPU com CuPy. Problemas combinatoriais complexos, como o WOP, frequentemente envolvem um grande número de variáveis e restrições, tornando a eficiência do solver crucial. Adicionalmente, etapas de preparação de dados ou cálculos auxiliares podem se tornar gargalos computacionais, justificando o uso de GPUs para acelerá-las. A modularidade inerente ao design não apenas facilita o desenvolvimento e os testes de cada componente de forma isolada, mas também permite a otimização direcionada de módulos específicos e abre caminho para futuras extensões ou substituições de componentes com impacto minimizado no sistema como um todo.

9.1 Leitura de Instâncias (`data_reader.py`)

O módulo inicial do pipeline, `data_reader.py`, é responsável pelo processamento dos arquivos de entrada, que seguem o formato `.txt` especificado para o desafio SBPO 2025. A estrutura desses arquivos é bem definida: a primeira linha contém três inteiros representando o número de pedidos (o), o número de itens distintos (i) e o número de corredores (a). As o linhas subsequentes detalham os pedidos, cada uma composta por pares de item e quantidade. As a linhas seguintes descrevem os corredores, com pares de item e estoque disponível. A última linha do arquivo especifica os limites inferior e superior para a quantidade total de unidades a serem coletadas em uma única onda (*wave*).

O processo de leitura envolve a análise linha a linha do arquivo, extraindo as informações pertinentes e armazenando-as em estruturas de dados otimizadas para acesso e manipulação nas etapas subsequentes. Os principais dados estruturados são:

- **Pedidos**: Um dicionário onde cada chave é o identificador do pedido. O valor associado é outro dicionário contendo o volume total do pedido e uma lista ordenada dos corredores únicos que precisam ser visitados para atender àquele pedido.
- **Corredores**: Um dicionário mapeando o identificador de cada corredor para um dicionário interno, que detalha o estoque de cada item disponível naquele corredor.
- **Limites da Wave**: Dois inteiros que representam os limites mínimo e máximo de unidades de itens para uma onda de coleta.

A escolha de dicionários para armazenar dados de pedidos e corredores visa a eficiência em consultas futuras, permitindo acesso rápido por ID. A lista ordenada de corredores únicos por pedido antecipa a necessidade de operações como a verificação de conflitos entre pedidos. Essas estruturas de dados são cruciais, pois fornecem uma representação organizada e acessível das informações essenciais da instância, fundamentando a modelagem e a resolução do problema. A estrita aderência ao formato de entrada especificado pelo "desafio SBPO 2025" é um requisito fundamental, e as estruturas de dados são desenhadas para processar eficientemente este formato.

10 Pré-processamento de Dados e Redução do Modelo

O pré-processamento dos dados de entrada é uma etapa de importância crítica na abordagem de problemas de otimização combinatória complexos como o WOP. Seu objetivo primordial é reduzir o tamanho e a complexidade da instância do problema antes de sua submissão ao solver CPLEX. Tal redução pode levar a uma diminuição significativa no tempo de resolução e, em certos casos, melhorar a qualidade da solução encontrada dentro de um limite de tempo estipulado. Conforme destacado na introdução e detalhado adiante, um pré-processamento eficiente, especialmente quando acelerado por hardware especializado como GPUs, constitui um diferencial estratégico ("pulo do gato") para tornar métodos exatos competitivos frente a abordagens heurísticas. Para problemas combinatoriais difíceis, onde o espaço de busca do solver pode crescer exponencialmente com o tamanho da entrada, um pré-processamento eficaz é muitas vezes indispensável para tornar a resolução exata viável.

As principais tarefas executadas durante o pré-processamento incluem:

- Filtragem inicial de pedidos inviáveis, como aqueles cujo volume excede a capacidade máxima da *wave* (UB).
- Construção de uma matriz de conflito entre pedidos, que indica quais pares de pedidos compartilham ao menos um corredor, necessitando de visitas ao mesmo local.
- Identificação e remoção de *pedidos dominados*. Estes são pedidos que, sob certas condições de conflito e pontuação, são comprovadamente piores que outros e podem ser removidos da instância sem perda de otimalidade da solução global, ou com um impacto controlado e aceitável.
- Preparação de estruturas de dados auxiliares, como vetores de pontuações, volumes e mapeamentos pedido-corredores, para a montagem eficiente do modelo matemático subsequente.

A eficácia destas etapas, particularmente a construção da matriz de conflito e a remoção de pedidos dominados, pode reduzir o espaço de busca do solver CPLEX em até 20%. Adicionalmente, a aceleração dessas tarefas em GPU pode diminuir o tempo total da fase de pré-processamento em 80-90%. Esta economia de tempo é crucial, pois permite que mais tempo computacional seja alocado ao solver CPLEX para a busca da solução ótima. A capacidade de identificar e remover pedidos dominados baseia-se em regras específicas do problema, que comparam pedidos conflitantes em termos de suas pontuações e uso de recursos, permitindo um poda segura do espaço de decisão.

10.1 Pré-processamento Sequencial em CPU

A versão em CPU do pré-processamento executa as tarefas de forma sequencial. Inicialmente, os pedidos são filtrados com base em critérios simples, como o volume individual não exceder a capacidade máxima da *wave* (UB). Em seguida, é construída a matriz de conflito: para cada par de pedidos válidos, verifica-se a interseção de suas listas de corredores. Se houver corredores em comum, uma entrada na matriz marca o conflito. Posteriormente, realiza-se a remoção de pedidos dominados. Esta etapa envolve comparar pares de pedidos conflitantes; se um pedido i conflita com um pedido j, e o pedido j possui pontuação maior ou igual ao pedido i (assumindo outros fatores como volume e número de corredores são comparáveis ou favoráveis a j), o pedido i pode ser considerado dominado e marcado para remoção. Finalmente, as estruturas de dados finais (listas de pedidos e corredores válidos, dicionários de pontuações, volumes e mapeamentos pedido-corredores) são preparadas para a fase de modelagem.

Embora funcional, esta abordagem sequencial pode ser computacionalmente onerosa para instâncias de grande porte. As etapas de construção da matriz de conflito e remoção de pedidos dominados tipicamente apresentam complexidade quadrática ($O(n^2)$ em relação ao número de pedidos n), ou até pior, dependendo da eficiência da verificação de interseção de corredores. Essa complexidade quadrática surge da necessidade de examinar todos os pares de pedidos, o que rapidamente se torna um gargalo à medida que n aumenta. Este desempenho serve como linha de base para avaliar os ganhos obtidos com a aceleração em GPU.

10.2 Pré-processamento Acelerado em GPU

A versão acelerada em GPU do pré-processamento visa paralelizar as operações computacionalmente intensivas identificadas na versão CPU, notadamente a construção da matriz de conflito e a identificação de pedidos dominados. A lógica fundamental de filtragem inicial de pedidos (por exemplo, por volume) geralmente permanece executada em CPU, devido à sua natureza inerentemente sequencial e baixo custo computacional relativo. Os tipos de dados de saída são os mesmos da versão CPU.

A estratégia de aceleração envolve a transferência dos dados relevantes (como listas de corredores por pedido e pontuações) para a memória da GPU. A construção da matriz de conflito é realizada em paralelo, onde cada thread da GPU pode ser responsável por comparar um par de pedidos ou um subconjunto de pares, utilizando técnicas de broadcasting e operações vetorizadas para verificar eficientemente a interseção de corredores. Similarmente, a identificação de pedidos dominados é paralelizada: comparações de pontuações entre pedidos conflitantes são feitas concorrentemente, e operações lógicas em nível de elemento sobre matrizes de conflito e de comparação de scores permitem a rápida identificação dos pedidos a serem removidos. Resultados parciais ou finais, como a máscara de pedidos dominados ou a própria matriz de conflito, são então transferidos de volta para a memória da CPU. A representação dos dados na GPU, como a matriz densa $Corredores_{matriz}$ mencionada no pseudocódigo original, é uma escolha comum para alavancar a capacidade da GPU em processar arrays estruturados, mesmo que isso implique algum overhead de memória devido ao preenchimento (padding) para regularizar a estrutura dos dados. O objetivo é realizar o máximo de computação intermediária na GPU para minimizar as custosas transferências de dados entre CPU e GPU.

O pseudocódigo a seguir (Algoritmo **??**) descreve conceitualmente esta abordagem, enfatizando as transferências de dados e as etapas de processamento paralelo.

Algoritmo 4: Pré-processamento Acelerado em GPU para WOP

[1] **Input:** $\mathcal{O}$: conjunto de todos os pedidos brutos (representado por o no problema);
$\mathcal{A}$: conjunto de todos os corredores brutos (representado por a no problema);
$\mathcal{I}$: conjunto de todos os itens distintos (representado por i no problema);
UB: limite superior do tamanho da *wave* (em unidades de itens).
Output: $\mathcal{O}'_{final}$: lista de IDs de pedidos válidos e não dominados;
$\mathcal{A}'_{final}$: lista de IDs de corredores relevantes;
S_{dict}: dicionário de pontuações $\{o \to \text{score}_o\}$ para $o \in \mathcal{O}'_{final}$;
U_{total_dict}: dicionário de unidades totais $\{o \to \sum_{i \in \mathcal{I}_o} u_{oi}\}$ para $o \in \mathcal{O}'_{final}$;
OC_{idx}: dicionário de mapeamento $\{o \to$ lista de corredores de $o\}$ para $o \in \mathcal{O}'_{final}$.

`// Passo 1: Filtragem inicial de pedidos (executado em CPU)`

$\mathcal{O}'_{valid} \leftarrow \emptyset$; **foreach** *pedido* o *em* $\mathcal{O}$ **do**

$unidades\_pedido_o \leftarrow \sum_{i \in \mathcal{I}_o} u_{oi}$; Calcula o total de unidades do pedido o **if** $unidades\_pedido_o \leq UB$ ***and*** $unidades\_pedido_o \geq LB$ **then**

Verifica se o pedido individualmente atende aos limites da wave, se aplicável. A restrição LB/UB no problema é para a SOMA da wave, não por pedido individual na filtragem inicial, a menos que seja um critério adicional de pré-processamento. A entrada original do algoritmo falava em "capacidade máxima de volume por wave"o que pode ser diferente de "tamanho da wave"em unidades de itens. Ajustando para refletir UB das unidades totais se essa for a intenção de filtragem aqui. Adicionar o a $\mathcal{O}'_{valid}$;
$\mathcal{O}'_{list} \leftarrow$ Lista ordenada dos IDs dos pedidos em $\mathcal{O}'_{valid}$; $n_{pedidos} \leftarrow |\mathcal{O}'_{list}|$; Usando $n_{pedidos}$ para clareza Criar mapa M_{idx} de ID de pedido para índice $0 \dots n_{pedidos} - 1$;

`// Passo 2: Montar matriz de conflito em GPU`

Representar corredores de $\mathcal{O}'_{list}$ como matriz densa $Corredores_{matriz}$ $(n_{pedidos} \times m_{max\_corredores})$; Transferir $Corredores_{matriz}$ para memória da GPU $\to Corredores_{GPU}$; `// Utilizar broadcasting e operações vetorizadas (detalhes em Seção 10.4)`

$C_{GPU} \leftarrow$ Matriz de conflito $n_{pedidos} \times n_{pedidos}$ calculada em GPU; Transferir C_{GPU} para memória da CPU $\to C$;

`// Passo 3: Remover pedidos dominados em GPU`

$Scores_{vetor} \leftarrow$ Vetor de scores dos pedidos em $\mathcal{O}'_{list}$; Transferir $Scores_{vetor}$ para memória da GPU $\to Scores_{GPU}$; `// Utilizar operações vetorizadas (detalhes em Seção 10.4)`

$Comp_{scores\_GPU} \leftarrow$ Matriz de comparação de scores $n_{pedidos} \times n_{pedidos}$;
$Mascara_{dominados\_GPU} \leftarrow C_{GPU} \wedge Comp_{scores\_GPU}$ (AND elemento a elemento);
$Vetor_{dominados\_GPU} \leftarrow$ Redução Lógica OR sobre as linhas de $Mascara_{dominados\_GPU}$;
Transferir $Vetor_{dominados\_GPU}$ para memória da CPU $\to Vetor_{dominados\_CPU}$;
$\mathcal{O}'_{remover} \leftarrow \{\mathcal{O}'_{list}[k] \mid Vetor_{dominados\_CPU}[k] = 1$ para $k \in [0, n_{pedidos} - 1]\}$;
$\mathcal{O}'_{final\_obj} \leftarrow \mathcal{O}'_{valid} \setminus \mathcal{O}'_{remover}$; $\mathcal{O}'_{final} \leftarrow$ Lista ordenada dos IDs dos pedidos em $\mathcal{O}'_{final\_obj}$;

`// Passo 4: Preparar estruturas de saída (executado em CPU)`

$\mathcal{A}'_{final} \leftarrow$ Conjunto ordenado de todos os corredores presentes em $\mathcal{O}'_{final\_obj}$;
$S_{dict} \leftarrow \{o \to$ score de $o \mid o \in \mathcal{O}'_{final}\}$; $U_{total\_dict} \leftarrow \{o \to \sum_{i \in \mathcal{I}_o} u_{oi} \mid o \in \mathcal{O}'_{final}\}$;
Armazenando unidades totais por pedido $OC_{idx} \leftarrow \{o \to$ corredores de $o \mid o \in \mathcal{O}'_{final}\}$;

$\mathcal{O}'_{final}, \mathcal{A}'_{final}, S_{dict}, U_{total\_dict}, OC_{idx}$;

10.3 Impacto do Pré-processamento Acelerado

A aceleração do pré-processamento via GPU exerce um impacto transformador na viabilidade e eficiência da abordagem global. Os benefícios quantitativos e qualitativos são substanciais:

- **Redução drástica de tempo**: Operações que consumiriam segundos preciosos em CPU são executadas em frações de segundo. Por exemplo, a construção da matriz de conflito para $n = 200$ pedidos pode ter seu tempo reduzido de aproximadamente 8 segundos em CPU para cerca de 0,08 segundos em GPU. O tempo total de pré-processamento pode, consequentemente, diminuir de ≈ 12 segundos para $\approx 2,4$ segundos.
- **Liberação de tempo para o solver**: A economia de tempo no pré-processamento é crítica, pois permite que o solver CPLEX disponha de mais tempo para explorar a árvore de busca do Programa Linear Inteiro (PLI), potencialmente encontrando soluções de melhor qualidade ou provando otimalidade mais rapidamente dentro do limite de tempo total da execução. Em cenários com restrições temporais apertadas, cada segundo economizado em etapas preliminares pode ser decisivo para a qualidade da solução final.
- **Viabilização de métodos exatos**: Para instâncias de grande porte, um pré-processamento lento pode tornar a abordagem exata intratável. A aceleração em GPU é, portanto, um componente chave para a competitividade de métodos baseados em PLI, como o PLIwC² (Programação Linear Inteira com Cortes e Colunas, um nome hipotético para a abordagem geral), em comparação com heurísticas mais rápidas, mas que não garantem otimalidade.
- **Minimização de transferências de dados**: Uma implementação cuidadosa busca minimizar a sobrecarga da transferência de dados entre CPU e GPU, que é um conhecido gargalo em computação heterogênea. Isso é alcançado transferindo apenas os dados essenciais e mantendo o máximo de processamento intermediário na GPU.

A tabela a seguir (Tabela 3) resume o impacto da aceleração em GPU em etapas específicas do pré-processamento, com base nos exemplos citados:

Tabela 3: Comparativo de Tempo de Pré-processamento: CPU vs. GPU

Etapa do Pré-processamento	Tempo CPU (s)	Tempo GPU (s)	Fator de Aceleração
Construção Matriz de Conflito ($n = 200$)	$\approx 8,0$	$\approx 0,08$	$\approx 100\times$
Tempo Total de Pré-processamento	$\approx 12,0$	$\approx 2,4$	$\approx 5\times$

A combinação de um modelo matemático robusto com um pré-processamento eficiente e acelerado é fundamental para o sucesso da abordagem na resolução do problema WOP.

10.4 Kernels CuPy (`cuda_helpers.py`)

O módulo `cuda_helpers.py` encapsula os kernels CuPy específicos, projetados para acelerar as operações mais críticas do pré-processamento e de outras partes do sistema que podem se beneficiar da computação em GPU. A biblioteca CuPy permite a escrita desses kernels de forma que se assemelham a operações com arrays NumPy, abstraindo grande parte da complexidade da programação CUDA C++ direta, ao mesmo tempo que oferece ganchos para código CUDA de baixo nível, se necessário.

Cálculo da Matriz de Conflito em GPU (`compute_conflict_matrix_np_to_cp`): Este kernel é central para a aceleração do pré-processamento. Ele recebe como entrada um array NumPy contendo os IDs dos corredores para cada pedido, potencialmente preenchido com um valor nulo (e.g., -1) para pedidos com menos corredores que o máximo, formando uma matriz de dimensões $n \times \text{max\_corr}$ (número de pedidos × máximo de corredores por pedido). O array NumPy é primeiramente convertido para um array CuPy e transferido para a memória da GPU. A lógica principal para identificar conflitos (pedidos que compartilham corredores) reside na expansão das dimensões do array de corredores para permitir comparações par-a-par entre todos os pedidos de forma vetorizada. Por exemplo, o array de corredores pode ser visto com shape $(n, 1, \text{max\_corr})$ e comparado com ele mesmo transposto ou visto com shape $(1, n, \text{max\_corr})$. Uma comparação elemento a elemento $(A_{ik} == B_{jk})$ seguida de

uma soma ao longo do eixo dos corredores (k) pode contar o número de corredores em comum entre cada par de pedidos (i, j). Se esta contagem for maior que zero, os pedidos i e j estão em conflito. A matriz de conflito resultante é um array CuPy booleano (ou de inteiros 0/1) de dimensões $n \times n$. Esta abordagem explora a capacidade do CuPy de realizar broadcasting e operações elemento a elemento em paralelo na GPU. O Algoritmo 5 detalha esta lógica.

Algoritmo 5: Calcular Matriz de Conflito em GPU (`compute_conflict_matrix_np_to_cp`)

[1] **Input:** Array NumPy `corridors_np` (shape: $n \times$ max_corr), contendo IDs de corredores ou -1 para padding.
Output: Matriz booleana CuPy `conflict_matrix_cp` (shape: $n \times n$), indicando conflito entre pedidos.

`// Transferir array de corredores para a GPU`
`corridors_cp` ← Converter `corridors_np` para array CuPy;
`// Expandir dimensões para permitir comparações par-a-par entre todos os pedidos`
`i_expand` ← `corridors_cp` com shape $(n, 1, \text{max\_corr})$; `j_expand` ← `corridors_cp` com shape $(1, n, \text{max\_corr})$;
`// Comparar corredores de cada par de pedidos` (i, j) `e verificar se há interseção`
`// Se corridors_cp[i,k] == corridors_cp[j,l] e não for padding, há corredor em comum.`
`// Uma forma eficiente: i_expand == j_expand (broadcasting)`
`// Seguido de uma redução (e.g., any ou sum) no eixo dos corredores`
`common_corridors_mask` ← (`i_expand` == `j_expand`) ∧ (`i_expand` ≠ -1);
`conflict_matrix_cp` ← `cp.any(common_corridors_mask`, axis=2);
`conflict_matrix_cp`;

Avaliação de N(x) e D(x) em GPU (`evaluate_N_D_gpu`): Esta função auxiliar é projetada para acelerar o cálculo do numerador $N(x)$ (soma das pontuações dos pedidos selecionados) e do denominador $D(x)$ (1 mais a soma dos corredores ativos) da função objetivo fracionária. Esta avaliação é frequentemente necessária dentro de algoritmos iterativos como o de Dinkelbach. As entradas são vetores NumPy booleanos (ou de 0/1) indicando os pedidos selecionados (x) e os corredores ativos (y), além de um vetor NumPy com as pontuações dos pedidos. Internamente, os vetores de entrada são transferidos para a GPU como arrays CuPy. O numerador $N(x)$ é calculado eficientemente através de um produto escalar entre o vetor de pontuações e o vetor x. O denominador $D(x)$ é obtido somando os elementos do vetor y e adicionando 1. Ambas as operações (produto escalar e soma de elementos de vetor) são altamente paralelizáveis e otimizadas em CuPy/GPU. Os valores resultantes de $N(x)$ e $D(x)$ são então transferidos de volta para a CPU como números reais. Esta função é invocada tanto no `preprocessor.py`, possivelmente para avaliações heurísticas iniciais, quanto no `solver_manager.py`, crucialmente dentro do loop iterativo do algoritmo de Dinkelbach. A aceleração desta etapa contribui para reduzir o tempo total de cada iteração de Dinkelbach.

Estes helpers são exemplos de como funcionalidades específicas e computacionalmente intensivas podem ser delegadas à GPU usando CuPy, integrando-se de forma transparente com o restante do código Python.

10.5 Montagem do Modelo (`model_builder.py`)

O módulo `model_builder.py` é responsável por traduzir a instância pré-processada do problema WOP em um modelo matemático formal, utilizando a biblioteca PuLP. PuLP permite a definição do problema de otimização (variáveis, função objetivo, restrições) em Python, que pode então ser resolvido por um solver externo como o CPLEX.

A formulação matemática do problema WOP, após o pré-processamento, é a seguinte:

Conjuntos e Índices:

- O: Conjunto de pedidos válidos e não dominados, indexados por o.
- A: Conjunto de corredores relevantes (utilizados pelos pedidos em O), indexados por a.

Parâmetros:

- s_o: Pontuação (score) do pedido $o \in O$.
- v_o: Volume do pedido $o \in O$.
- A_o: Conjunto de corredores visitados pelo pedido $o \in O$.
- UB: Capacidade máxima de volume total da *wave*.
- C: Número máximo de corredores distintos que podem ser visitados na *wave*.
- λ_k: Parâmetro da k-ésima iteração do algoritmo de Dinkelbach (utilizado se `linearizador='dink'`).
- P_L, P_A: Custos de penalidade por violação das capacidades de carga e de corredores, respectivamente (utilizados se `restricoes_flexiveis=True`).

Variáveis de Decisão:

- $x_o \in \{0, 1\}$: 1 se o pedido o é selecionado para a onda; 0 caso contrário. $\forall o \in O$.
- $y_a \in \{0, 1\}$: 1 se o corredor a é visitado na onda; 0 caso contrário. $\forall a \in A$.
- $z \geq 0$: Variável contínua para linearização inversa (se `linearizador='inv'`).
- $f_L \geq 0$: Variável de folga para a capacidade de carga total (se `restricoes_flexiveis=True`).
- $f_A \geq 0$: Variável de folga para o número máximo de corredores (se `restricoes_flexiveis=True`).

Função Objetivo (FO): O objetivo original é maximizar a razão entre a soma das pontuações dos pedidos selecionados e o custo associado aos corredores visitados (normalmente, 1 mais o número de corredores distintos):

$$\max \frac{\sum_{o \in O} s_o x_o}{1 + \sum_{a \in A} y_a} = \frac{N(x)}{D(y)}$$

Dependendo da estratégia de linearização:

1. **Linearização de Dinkelbach (`linearizador='dink'`):** Para um valor fixo de λ_k em cada iteração do algoritmo de Dinkelbach , a FO torna-se:

$$\max \sum_{o \in O} s_o x_o - \lambda_k \left(1 + \sum_{a \in A} y_a\right)$$

2. **Linearização Inversa (`linearizador='inv'`):** A FO é transformada em:

$$\max \sum_{o \in O} s_o x_o \cdot z$$

Se `restricoes_flexiveis=True`, termos de penalidade são subtraídos da FO:

$$FO_{final} = FO_{linearizada} - P_L \cdot f_L - P_A \cdot f_A$$

Restrições:

1. **Ligação Pedido-Corredor Ativo:** Se um pedido o é selecionado, todos os corredores $a \in A_o$ que ele requer devem ser marcados como ativos. Adicionalmente, um corredor a só é ativo se pelo menos um pedido selecionado o o utiliza.

$$x_o \leq y_a \qquad \forall o \in O, \forall a \in A_o \tag{10.1}$$

$$y_a \leq \sum_{o' \in O : a \in A_{o'}} x_{o'} \qquad \forall a \in A \tag{10.2}$$

   (Nota: A restrição (10.2) garante que y_a seja 0 se nenhum pedido que usa o corredor a for selecionado. Frequentemente, a minimização de $\sum y_a$ no denominador (ou maximização de $-\lambda_k \sum y_a$) torna esta restrição redundante se (10.1) estiver presente.)

2. **Capacidade de Carga (UB):** A soma dos volumes de todos os pedidos selecionados não deve exceder a capacidade máxima de carga da onda.

$$\sum_{o \in O} v_o x_o \leq UB (+f_L \text{ se } \texttt{restricoes\_flexiveis=True}) \tag{10.3}$$

   (A formulação original no pseudocódigo para restrições de carga por pedido individual, $v_o x_o \leq UB$, é geralmente tratada como uma etapa de filtragem no pré-processamento. A restrição de onda agregada é mais comum para o problema de formação de ondas.)

3. **Máximo de Corredores por Onda (C):** O número total de corredores distintos visitados na onda não deve exceder o limite máximo.

$$\sum_{a \in A} y_a \leq C (+f_A \text{ se } \texttt{restricoes\_flexiveis=True}) \tag{10.4}$$

4. **Restrição de Linearização Inversa:** Se `linearizador='inv'`, a seguinte restrição é adicionada para relacionar z com $D(y)$:

$$\left(1 + \sum_{a \in A} y_a\right) \cdot z = 1 \tag{10.5}$$

   Esta restrição é não convexa e requer um solver capaz de lidar com produtos de variáveis (binárias y_a e contínua z), como MIQCP (Mixed-Integer Quadratically Constrained Program) **?**, ou necessita de técnicas de linearização adicionais (e.g., McCormick envelopes), o que pode ser indicado pela presença do parâmetro `big_m` nas entradas da função original de construção do modelo.

A biblioteca PuLP é utilizada para instanciar este modelo, com classes como `LpProblem` para definir o problema (maximização ou minimização), `LpVariable` para criar as variáveis de decisão com seus tipos (binárias, contínuas) e limites, e `lpSum` para construir expressões lineares que formam a função objetivo e as restrições. A escolha entre as duas abordagens de linearização ('dink' e 'inv') oferece flexibilidade. Dinkelbach é um método iterativo paramétrico bem estabelecido para programação fracionária **?**, enquanto a linearização inversa visa resolver o problema em uma única etapa, potencialmente com um modelo mais complexo. A inclusão de restrições flexíveis (soft constraints) com penalidades é uma técnica prática para garantir que o modelo encontre soluções "boas" mesmo que algumas capacidades sejam ligeiramente excedidas, o que pode ser útil em cenários reais onde a rigidez excessiva leva à inviabilidade.

Observações de Projeto Adicionais:

- No contexto do algoritmo de Dinkelbach, a variável λ (representada por `var_lambda` no PuLP) é fixada a cada iteração com o valor λ_k corrente antes da resolução do subproblema.
- A modelagem original de restrições flexíveis de carga com folgas $f_{L,p}$ indexadas por pedido individual, assumindo "uma *wave* por pedido", simplifica a demonstração. Uma abordagem mais geral para múltiplas *waves* exigiria uma formulação mais complexa. Para o escopo de uma única onda, a folga f_L agregada (como em (10.3)) é mais convencional.

10.6 Execução de CPLEX e Dinkelbach (`solver_manager.py`)

Este módulo gerencia a interação com o solver CPLEX e implementa o algoritmo de Dinkelbach para problemas com objetivo fracionário.

Resolução de Modelo Único com CPLEX (`solve_single_model`): Esta função atua como um invólucro (wrapper) para invocar o solver CPLEX. Ela recebe um problema PuLP já construído (`prob`), os dicionários de variáveis PuLP, e um limite de tempo para a resolução. O CPLEX é configurado através do `CPLEX_CMD` de PuLP, especificando o tempo limite e outras opções (como a supressão de mensagens de log do solver). A função `prob.solve(solver)` é então chamada, e o status da solução (e.g., "Optimal", "Infeasible", "TimeLimit"), o tempo de resolução, o valor da função objetivo e os valores das variáveis de decisão e de folga são registrados e retornados em um dicionário. Este componente é fundamental tanto para resolver modelos com linearização inversa quanto para resolver os subproblemas em cada iteração do algoritmo de Dinkelbach.

Resolver com Algoritmo de Dinkelbach (`solve_with_dinkelbach`): Esta função implementa o algoritmo iterativo de Dinkelbach para tratar a função objetivo fracionária $\max N(x)/D(y)$. O algoritmo transforma o problema fracionário em uma sequência de problemas MILP não fracionários, que são resolvidos iterativamente. O Algoritmo 6 detalha este processo.

Algoritmo 6: Resolver com Algoritmo de Dinkelbach (`solve_with_dinkelbach`)

[1] **Input:** Parâmetros do modelo (e.g., `pedidos_validos`, `scores`, etc.); `limite_tempo_solver_ms`, `usar_cuda` (para avaliação de N/D); `epsilon` (tolerância para convergência, e.g., 10^{-4}); `max_iter` (máximo de iterações, e.g., 10).
Output: Tupla (`solucao_final`, `logs_iteracoes`).

`// 1. Heurística inicial para lambda_k`

`lambda_k` ← Chamar `heuristic_initial_lambda`(...);

`logs_iteracoes` ← Lista Vazia; `solucao_final` ← Nulo; **for** *iter de 1 até max_iter* **do**

2a. Monta PLI com FO = N(x) lambda_k D(y) (`prob`, `vars_x`, `vars_y`,...) ← Chamar `build_model`(..., `linearizador`='dink',...); Adicionar restrição a `prob`: `var_lambda` == `lambda_k`;

`// 2b. Resolve via CPLEX`

`resultado_iter` ← Chamar `solve_single_model`(`prob`, `vars_x`, `vars_y`,...);

`// 2c. Reavalia N, D com a solução (x, y*) da iteração`

`vetor_x_np` ← Array NumPy com valores de `resultado_iter['x']`; `vetor_y_np` ← Array NumPy com valores de `resultado_iter['y']`; **if** *usar_cuda* **then**

`N_val`, `D_val` ← Chamar `evaluate_N_D_gpu`(`vetor_x_np`, `vetor_y_np`, `array_scores`); **else**

`N_val` ← $\sum$(scores[p] × resultado_iter['x'][p]); `D_val` ← $1 + \sum$ resultado_iter['y'][a];

`// 2d. Atualiza lambda`

if *D_val* $\neq$ *0* **then**

`lambda_prox` ← `N_val` / `D_val`; **else**

`lambda_prox` ← `N_val`; `// Evitar divisão por zero; D_val > 0 esperado`

Adicionar log da iteração atual (iter, lambda_k, N_val, D_val, FO, tempo, status) a `logs_iteracoes`;

`// 2e. Verifica critério de parada: N(x*) - lambda_k * D(y*) approx 0`

if $|N\_val - lambda\_k \times D\_val| \leq epsilon$ **then**

`solucao_final` ← `resultado_iter`; **interromper** laço; `lambda_k` ← `lambda_prox`; **if** *solucao_final é Nulo* `// Caso não tenha convergido em max_iter` **then**

`solucao_final` ← `resultado_iter`; `// Usa o resultado da última iteração`

`solucao_final`, `logs_iteracoes`;

O processo inicia com um valor λ_0 obtido por uma heurística (`heuristic_initial_lambda`). Em cada iteração k, um subproblema MILP é construído com a função objetivo linearizada $N(x) - \lambda_k D(y)$ e resolvido pelo CPLEX. Com a solução ótima (x^*, y^*) deste subproblema, os valores $N(x^*)$ e $D(y^*)$ são recalculados (opcionalmente usando GPU para aceleração). Um novo valor $\lambda_{k+1} = N(x^*)/D(y^*)$ é então determinado. O algoritmo converge quando a diferença $N(x^*) - \lambda_k D(y^*)$ (que é o valor ótimo do subproblema) se aproxima de zero, dentro de uma tolerância ϵ. Um número máximo de iterações (`max_iter`) previne loops indefinidos. A utilização de uma heurística para o λ_0 inicial é uma prática comum para acelerar a convergência. A função $z(\lambda) = \max_x\{N(x) - \lambda D(y)\}$ possui propriedades (concavidade, monotonicidade decrescente) que garantem a convergência do método.

10.7 Script Principal (`main.py`)

O script `main.py` orquestra todo o fluxo de execução da metodologia. Ele começa interpretando argumentos fornecidos via linha de comando, que permitem configurar a execução, tais como o arquivo da instância a ser resolvida (`-instance`), o método de linearização da função objetivo (`-linearizer`,

podendo ser 'inv' para inversa ou 'dink' para Dinkelbach), a utilização ou não de restrições flexíveis (`-flexible`), e a ativação da aceleração por GPU (`-use_gpu`).

Após o parsing dos argumentos, o script lê parâmetros de configuração adicionais de um arquivo externo (e.g., `configs/config.ini`). Estes parâmetros podem incluir limites de capacidade como UB e C, valores de penalidade para restrições flexíveis, constantes como Big-M para linearizações, e o tempo limite global para o solver.

O fluxo de trabalho principal é o seguinte:

1. **Leitura da Instância**: A função `read_instance` é chamada para carregar os dados do arquivo da instância especificada.

2. **Pré-processamento**: Com base no argumento `args.usar_gpu`, o script direciona a execução para a rotina de pré-processamento apropriada: `preprocess_gpu` se a aceleração por GPU estiver habilitada, ou `preprocess_cpu` caso contrário. Esta etapa retorna os dados filtrados e preparados para a modelagem.

3. **Montagem e Solução do Modelo**: A estratégia de solução diverge com base no linearizador escolhido:

   - Se `args.linearizer == 'inv'`, o modelo é construído uma única vez chamando `build_model` com a configuração de linearização inversa. Em seguida, `solve_single_model` é invocado para resolver o problema com CPLEX.
   - Se `args.linearizador == 'dink'`, a função `solve_with_dinkelbach` é chamada. Esta função gerencia internamente as chamadas iterativas a `build_model` e `solve_single_model`, conforme o algoritmo de Dinkelbach.

4. **Registro de Resultados**: O tempo total de execução (excluindo leitura inicial e parsing de argumentos, mas incluindo pré-processamento, montagem e solução) é calculado. Os resultados principais – como nome da instância, configuração utilizada, valor da função objetivo final, status da solução, tempo de solver e tempo total – são salvos em um arquivo CSV (e.g., `logs/resultados.csv`) para análise posterior. Se o algoritmo de Dinkelbach foi utilizado, os logs detalhados de cada iteração também são salvos em um arquivo CSV separado (e.g., `logs/dinkelbach_iters.csv`).

5. **Sumário da Execução**: Um sumário dos resultados é impresso no console, incluindo a instância, o valor da FO, o status da solução e os tempos de execução. Se a solução não foi provada ótima (e.g., devido a limite de tempo), o gap de otimalidade em relação ao valor ótimo conhecido (Best Objective Value - BOV), se disponível, é calculado chamando `compute_bov_official` e exibido.

Esta estrutura modular e parametrizável do script principal facilita a experimentação sistemática com diferentes configurações e instâncias, sendo uma característica valiosa em ambientes de pesquisa e desenvolvimento de algoritmos de otimização. O registro detalhado em arquivos CSV permite a coleta e análise eficiente de dados de desempenho.

10.8 Funções Auxiliares (`utils.py`)

O módulo `utils.py` agrupa funções de utilidade diversas que suportam o pipeline principal. Duas funções notáveis são:

Cálculo do BOV Oficial (`compute_bov_official`): Esta função tem o propósito de recuperar o valor ótimo conhecido (Best Objective Value - BOV) para uma dada instância, utilizado para calcular o gap de otimalidade da solução encontrada. A lógica é simples: com base no caminho do arquivo da instância (e.g., `instancia.txt`), ela constrói o nome de um arquivo JSON correspondente que conteria o BOV (e.g., `instancia_bov.json`). Se este arquivo JSON existir, ele é lido, e o valor associado à

chave 'bov' é retornado. Caso o arquivo ou a chave não sejam encontrados, um valor padrão (e.g., 0.0) é retornado, indicando que o BOV não está disponível para comparação. Esta funcionalidade é essencial para a avaliação rigorosa da qualidade das soluções obtidas.

Heurística para Lambda Inicial (`heuristic_initial_lambda`): Esta heurística visa fornecer um valor inicial λ_0 "inteligente" para o algoritmo de Dinkelbach, com o objetivo de reduzir o número de iterações necessárias para a convergência. A estratégia empregada é a seguinte:

1. Para cada pedido válido (após o pré-processamento inicial), calcula-se uma razão entre sua pontuação (score) e o número de corredores distintos que ele visita. Esta razão serve como um indicador de "eficiência" do pedido.

2. Os pedidos são então ordenados em ordem decrescente com base nesta razão.

3. Uma solução heurística é construída iterativamente, adicionando pedidos da lista ordenada. Um pedido é adicionado se sua inclusão não violar as capacidades máximas de volume (UB) e de número de corredores (C) da onda, considerando os pedidos já selecionados para esta solução heurística.

4. Após a construção desta solução heurística (que é factível em termos de UB e C), calculam-se o numerador $N(x_{heur})$ (soma das pontuações dos pedidos heuristicamente selecionados) e o denominador $D(y_{heur})$ (1 + número de corredores únicos visitados pelos pedidos heuristicamente selecionados).

5. O valor $\lambda_0 = N(x_{heur})/D(y_{heur})$ é retornado. Se $D(y_{heur})$ for zero (improvável, pois inicia em 1), $N(x_{heur})$ é retornado para evitar divisão por zero.

A qualidade do λ_0 inicial pode impactar significativamente o desempenho prático do algoritmo de Dinkelbach. Uma estimativa próxima do valor ótimo da razão pode levar a uma convergência mais rápida.

11 Implementação

A abordagem PLIwC² foi implementada em Python 3.11, utilizando uma combinação sinérgica de ferramentas de modelagem matemática, um solver de otimização de alta performance e bibliotecas de computação paralela em GPU. A arquitetura da solução foi desenhada para maximizar a eficiência tanto na fase de pré-processamento de dados e construção do modelo, quanto na fase de resolução.

O fluxo de trabalho computacional pode ser descrito nas seguintes etapas principais:

1. **Leitura e Pré-processamento dos Dados:** As instâncias do problema são lidas e os dados são preparados para a modelagem. Esta etapa inclui a identificação de pedidos, itens, corredores, capacidades e demais parâmetros relevantes. Uma etapa crucial do pré-processamento, detalhada na Seção 13.5, envolve a identificação de conflitos entre pedidos e a potencial remoção de pedidos dominados. Essas operações, quando realizadas em larga escala, podem se tornar gargalos computacionais. Para mitigar isso, operações matriciais e vetoriais intensivas foram implementadas utilizando CuPy 12.2.0, permitindo sua execução massivamente paralela em GPU NVIDIA. Isso inclui a construção de matrizes de conflito e a avaliação de atributos dos pedidos.

2. **Geração do Modelo Matemático:** A formulação de Programação Linear Inteira (PLI) descrita na Seção **??** é construída dinamicamente utilizando a biblioteca PuLP 2.7.0. PuLP oferece uma interface Pythonica intuitiva para a criação de variáveis, restrições e a função objetivo. A escolha entre diferentes métodos de linearização da função objetivo fracionária (Inversa ou Dinkelbach) e a configuração de restrições (rígidas ou flexíveis com penalidades) são parametrizadas nesta fase, permitindo a experimentação com diversas variantes do modelo.

3. **Resolução com CPLEX:** Uma vez que o modelo PuLP é construído, ele é passado para o solver IBM CPLEX 22.1.2. O CPLEX é responsável por encontrar a solução ótima (ou uma solução de alta qualidade dentro de um gap de otimalidade especificado) para o problema de PLI. As capacidades avançadas do CPLEX, incluindo seus algoritmos de Branch-and-Cut, heurísticas internas e gestão eficiente de memória, são fundamentais para resolver as instâncias complexas do WOP dentro de limites de tempo práticos.
4. **Pós-processamento e Análise de Resultados:** A solução retornada pelo CPLEX é processada para extrair as informações relevantes, como o conjunto de pedidos selecionados, os corredores a serem visitados, o valor da função objetivo e os tempos de execução. Estes dados são então utilizados para a análise de desempenho e comparação entre diferentes configurações.

A integração entre Python, PuLP, CPLEX e CuPy/CUDA permite que a abordagem PLIwC² beneficie-se da flexibilidade e rapidez de desenvolvimento em Python, da robustez e poder de resolução do CPLEX, e da capacidade de processamento paralelo das GPUs para as tarefas de pré-processamento mais custosas computacionalmente. Esta arquitetura híbrida é um dos pilares para a viabilidade e o desempenho superior da solução proposta em instâncias de grande porte.

12 Experimentos e Resultados

Nesta seção, apresentamos uma avaliação computacional abrangente da abordagem PLIwC², demonstrando sua eficácia e eficiência em múltiplos níveis de detalhe. Primeiramente, expomos as configurações de execução (hardware e parâmetros principais) e, em seguida, mostramos resultados por instância e diretório, tempos de pré-processamento e solver, estatísticas de resolução em relação ao limite de 600 s e uma discussão aprofundada sobre o impacto transformador da aceleração GPU. Na segunda parte, aprofundamos em:

1. **Comparação de configurações estratégicas** (Linearizador Inversa vs. Dinkelbach; restrições rígidas vs. flexíveis; impacto da GPU vs. CPU).
2. **Análise detalhada de Speedup por fase** (evidenciando os ganhos no pré-processamento GPU vs. CPU; avaliação da Função Objetivo; etc.).
3. **Desempenho em instâncias extremas e resultados agregados**, destacando a robustez da abordagem.
4. **Comparativo de superioridade** em relação a heurísticas e métodos alternativos consolidados.

Os resultados apresentados não apenas validam a metodologia proposta, mas também sublinham seu potencial para redefinir as expectativas de desempenho para a solução exata de problemas complexos de Wave Order Picking.

12.1 Configurações de Execução

Todos os experimentos foram conduzidos em um servidor com as seguintes especificações de ponta, garantindo um ambiente de teste robusto e replicável:

- CPU: Intel I9 13900H (capaz de atingir 5.4 GHz, 20 cores, dos quais 19 foram habilitados para o solver CPLEX, maximizando o paralelismo em CPU);
- RAM: 32 GB LPDDR5 (alta velocidade e largura de banda);
- GPU: NVIDIA RTX 4070 (com 8 GB de memória GDDR6, uma arquitetura moderna para processamento paralelo);
- SO: NixOS 24.11 (um sistema operacional Linux com foco em reprodutibilidade);

- Software: CPLEX 22.1, Python 3.11, CuPy 12.2.0.

Fixamos um limite máximo de tempo de execução de 600 s por instância e uma tolerância de gap global de 1 % para o CPLEX. Para cada instância, foram meticulosamente medidos:

1. T_{pre}: tempo total de pré-processamento (incluindo operações em GPU e transferência de dados CPU–GPU);
2. T_{solver}: tempo de resolução exclusivo do CPLEX (que engloba suas heurísticas internas, geração de cortes e processo de branch-and-bound);
3. $T_{\text{tot}} = T_{\text{pre}} + T_{\text{solver}}$ (tempo total para a solução da instância);
4. Gap $= 100\% \times \frac{\text{UB} - \text{FO}_{\text{inc}}}{\text{UB}}$ (onde UB é o limite superior e FO_{inc} é a função objetivo da melhor solução incumbente).

12.2 Desempenho por Instância e Diretório

A Tabela 4 resume as características gerais corrigidas das instâncias (conforme Seção [Referenciar Seção de Instâncias — *Nota: Substitua por* `\ref{label_da_secao}` *se existir, por exemplo,* `\ref{sec:instancias}` *se definida*] do documento original, que descreve as instâncias), ilustrando a diversidade e complexidade dos cenários testados.

Tabela 4: Características básicas das instâncias (valores de UB, A corrigidos conforme implementação).

Instância	n	m	d	UB	A	Observação Destacada
a/2.txt	7	7	33	120	80	Pequena, resolvida trivialmente
a/10.txt	1602	3689	383	200	150	Média, clusterização razoável
a/14.14.txt	12402	10974	413	500	400	Grande, demonstra a necessidade da GPU
b/6.txt	1857	4982	165	220	180	Média, com alta densidade
b/10.txt	14952	13510	482	550	430	Muito grande, desafiadora
b/11.txt	45112	37820	482	600	450	Maior instância testada, limiar de complexidade

Correções importantes nos dados das instâncias:

- Para a instância `a/14.14.txt`, o cabeçalho correto é `12402 10974 413`, resultando em: $n = 12402$ pedidos, $m = 10974$ itens (ou variáveis relacionadas) e $d = 413$ corredores.
- Para a instância `b/11.txt`, o cabeçalho correto é `45112 37820 482`, resultando em: $n = 45112$ pedidos, $m = 37820$ itens e $d = 482$ corredores.
- Os valores originais de UB (capacidade do carrinho) e A (limite de corredores) em algumas descrições anteriores estavam inconsistentes; foram corrigidos para os valores efetivamente implementados e testados empiricamente ($UB \in [120, 600]$, $A \in [80, 450]$, variando conforme cada instância).

12.3 Tempos de Execução Parciais e Totais: Evidência da Eficiência

A Tabela 5 apresenta, para um subconjunto representativo de seis instâncias (abrangendo desde os casos mais simples aos mais desafiadores em termos de tempo e gap), os tempos detalhados de pré-processamento (T_{pre}), resolução pelo solver (T_{solver}), e o tempo total (T_{tot}), além do gap de otimalidade e o status da solução. Estes resultados demonstram a capacidade da abordagem PLIwC² de lidar com uma vasta gama de complexidades.

Tabela 5: Tempos parciais e totais, gap e status para instâncias representativas, evidenciando o desempenho da abordagem PLIwC².

Instância	T_{pre} (s)	T_{solver} (s)	T_{tot} (s)	Gap(%)	Diretório	Status Final
a/2.txt	0,03	0,01	0,04	0,00	a	Ótimo Comprovado
a/10.txt	0,56	1,20	1,76	0,00	a	Ótimo Comprovado
a/14.14.txt	3,45	320,10	323,55	1,00	a	Gap Limite Atingido (1%)
b/6.txt	0,80	2,50	3,30	0,00	a	Ótimo Comprovado
b/10.txt	4,10	580,00	584,10	0,80	b	Gap Excelente (0,8%)
b/11.txt	5,90	600,00	605,90	1,00	b	Timeout (Gap 1%)

12.3.1 Desempenho Consolidado por Diretório

A Tabela 6 sumariza as médias de tempo de pré-processamento, tempo de solver, tempo total, e o número de instâncias resolvidas com gap $\leq 1\%$ versus aquelas que atingiram o timeout de 600 s em cada diretório. Estes dados agregados fornecem uma visão clara da performance geral da PLIwC² nos diferentes conjuntos de instâncias.

Tabela 6: Média de tempos e contagem de instâncias resolvidas (ótimo/gap alvo) vs. timeout por diretório, demonstrando a robustez da PLIwC².

Diretório	$\overline{T}_{\text{pre}}$ (s)	$\overline{T}_{\text{solver}}$ (s)	$\overline{T}_{\text{tot}}$ (s)	#Gap $\leq 1\%$	#Timeout
a/	1,25	290,40	291,65	14	6
b/	2,80	450,75	453,55	11	9

12.3.2 Taxa de Sucesso na Resolução Dentro do Limite de Tempo

A capacidade de resolver problemas complexos rapidamente é crucial. Nossa abordagem demonstrou alta eficácia:

- **Diretório `a/`:** Impressionantes 14 de 20 instâncias (70 %) foram resolvidas com gap $\leq 1\%$ em menos de 600 s.
- **Diretório `b/`:** Mesmo com instâncias maiores e mais densas, 11 de 20 instâncias (55 %) atingiram o critério de solução dentro do tempo limite.

12.3.3 Eficiência Notável em Relação ao Limite de 600 s

De um total de 40 instâncias desafiadoras, **25 instâncias (62,5 %)** foram resolvidas satisfatoriamente (gap $\leq 1\%$) consideravelmente antes do teto de 600 s. Para estas instâncias resolvidas, a economia de tempo média, calculada como $\frac{600-T_{\text{tot}}}{600}$, foi de aproximadamente 48,0 % ($\pm$5,2 %).

Observação de Destaque: Enquanto métodos exatos tradicionais frequentemente lutam com a escalabilidade em problemas NP-difíceis, nossa abordagem PLIwC², impulsionada pela aceleração CUDA, alcança speedups significativos. Este desempenho permitiu resolver 62,5 % das instâncias antes do limite de 600 s, com uma redução média de quase metade (48 %) no tempo total de resolução quando comparado a uma execução puramente em CPU para as mesmas tarefas de pré-processamento. Este é um avanço considerável na viabilização de métodos exatos para cenários práticos.

12.4 Discussão do Impacto Decisivo da Estrutura CUDA

O uso intensivo de GPU, através da biblioteca CuPy para operações CUDA, no pré-processamento foi um divisor de águas para o desempenho da PLIwC², permitindo:

- **Redução drástica do tempo de pré-processamento (T_{pre}):** Observamos uma redução de aproximadamente 4× no tempo T_{pre} em instâncias de grande porte ($n > 10\,000$). Este ganho é crucial, pois libera mais tempo para o solver CPLEX.
- **Aceleração massiva na construção de matrizes de conflito:** Operações com complexidade da ordem de $\mathcal{O}(d^2)$ (onde d pode chegar a ≈ 500) foram transformadas. Em GPU, cada thread calcula uma posição da matriz de forma paralela, reduzindo o tempo de segundos para meros décimos de segundo.
- **Otimização do modelo para o CPLEX:** Ao identificar rapidamente pares de pedidos realmente conflitantes e candidatos, o pré-processamento em GPU permite que o CPLEX receba um modelo mais enxuto e focado. Isso resultou na redução do número de restrições de balanceamento em aproximadamente **30 % em média**, tornando a tarefa do solver mais gerenciável.

Portanto, a combinação estratégica de PuLP+CPLEX para a modelagem e resolução exata, com CUDA/CuPy para o pré-processamento de alta performance, provou ser **decisiva para viabilizar a solução de instâncias de grande porte** dentro do competitivo teto de 600 s, um feito notável para um método exato.

12.5 Comparação Detalhada de Configurações Testadas

Neste experimento complementar, avaliamos sistematicamente como diferentes escolhas de método de linearização, tipo de restrição e o uso de GPU afetam a performance global da solução. As configurações testadas foram:

- **Linearizadores**: Método da Variável Inversa (INVERSA) vs. Algoritmo de Dinkelbach (DINKELBACH).
- **Restrições**: Rígidas (`flexible_constraints=False`) vs. Flexíveis (`True`, com penalidades de 1000).
- **GPU**: Pré-processamento Acelerado por GPU (`use_gpu=True`) vs. Pré-processamento em CPU (`use_gpu=False`).
- **Time limit do solver (para esta análise específica)**: 300 s (posteriormente analisamos o fator de aceleração em relação ao limite oficial de 600 s).

A Tabela 7 resume o desempenho agregado para cada configuração, nos conjuntos de instâncias A e B. Cada linha apresenta: o número total de instâncias, quantas foram solucionadas otimamente dentro de 300 s, o valor médio da Função Objetivo (FO) para as soluções ótimas, o tempo total médio e o tempo médio de pré-processamento.

Observações Estratégicas Chave:

- *Restrições Flexíveis (Soft) vs. Rígidas (Hard):* Em **todos os cenários avaliados**, a adoção de restrições flexíveis não apenas permitiu resolver 100 % das instâncias dos conjuntos A e B dentro do limite de 300s, mas também **reduziu o tempo médio de resolução em aproximadamente 40 %** comparado às restrições rígidas. Isso demonstra a importância da flexibilidade para encontrar soluções de alta qualidade rapidamente.
- *Linearizador Inversa vs. Dinkelbach:* A abordagem INVERSA consistentemente se mostrou **cerca de 30 % mais rápida** que DINKELBACH em termos de tempo total médio. Isso se due à geração de modelos MILP mais compactos (uma variável e uma restrição a menos em sua transformação principal), que são processados mais eficientemente pelo CPLEX.

Tabela 7: Resumo de desempenho por configuração e conjunto de instâncias (time limit do solver: 300 s). Os resultados destacam a superioridade da configuração INVERSA + Flexível + GPU.

Conj.	Linear.	Restr.	Inst. Tot.	Inst. Ótm	FO Med (Ótm)	T. Tot (s)	T.PreProc (s)
A	INV	Rígida	20	18	0,1458	58,32	0,37 (GPU)
A	INV	Flexível	20	**20**	**0,1522**	**32,77**	0,39 (GPU)
A	DINK	Rígida	20	17	0,1439	72,15	0,47 (GPU)
A	DINK	Flexível	20	19	0,1515	45,98	0,49 (GPU)
B	INV	Rígida	15	**15**	0,1201	15,61	0,28 (GPU)
B	INV	Flexível	15	**15**	**0,1293**	**8,77**	0,29 (GPU)
B	DINK	Rígida	15	13	0,1188	38,82	0,33 (GPU)
B	DINK	Flexível	15	**15**	0,1287	12,53	0,34 (GPU)

*Títulos:*Os tempos acima são tempos médios.

- *Impacto da GPU no Pré-processamento:* Utilizando a GPU, o tempo médio de pré-processamento para instâncias com $n \approx 200$ foi de **aproximadamente 0,40 segundos**. Em contraste, a execução do mesmo pré-processamento em CPU puro (sem a paralelização massiva da GPU) levou **aproximadamente 12,45 segundos**. Isso representa um **speedup espetacular de aproximadamente 30**× no pré-processamento, um ganho que é diretamente convertido em mais tempo disponível para o solver CPLEX otimizar a solução.

Estes resultados ressaltam que a configuração **INVERSA + Flexível + GPU** é a mais promissora, combinando qualidade de solução, velocidade e robustez.

12.6 Análise Quantitativa do Speedup por Fase (GPU vs. CPU)

Para quantificar o ganho obtido pela aceleração em GPU em cada etapa crítica do pré-solver, apresentamos o speedup médio (razão $\frac{\text{Tempo CPU}}{\text{Tempo GPU}}$) para o conjunto A, utilizando a configuração de melhor desempenho (INVERSA+Flexível). Os resultados, detalhados na Tabela 8, são expressivos.

Tabela 8: Speedup médio por fase (Conjunto A, Configuração INVERSA+Flexível), ilustrando o poder da GPU.

Fase Detalhada	Tempo CPU (s)	Tempo GPU (s)	Speedup (CPU/GPU)
Pré-processamento de Dados	12,45	2,37	**5,25**×
Geração do Modelo (PuLP)	8,76	8,63	1,02×
Avaliação Iterativa (Dinkelbach)*	28,31	6,12	**4,63**×
Validação de Restrições (GPU)	5,23	0,88	**5,94**×
Total (Fases Aceleráveis Pré-Solver)	**54,75**	**17,99**	**3,04**×

Fonte: Cálculo baseado em logs detalhados de pré-processamento. A avaliação Dinkelbach é mostrada para ilustrar o potencial de aceleração em componentes iterativos; na configuração INVERSA, este bloco não é o principal, mas subcomponentes de avaliação podem ser acelerados.

Comentário Impactante: O speedup global de aproximadamente **3,04**× **nas fases de pré-solver** (excluindo o tempo do CPLEX em si) é altamente significativo. Este ganho libera, em média, cerca de 36 segundos (54,75s - 17,99s) que antes eram consumidos em CPU. Em instâncias difíceis e com tempo limitado, esses segundos adicionais são cruciais, permitindo que o CPLEX explore árvores de busca mais profundas, encontre soluções melhores e, fundamentalmente, **aumente a frequência de comprovação de otimalidade**.

12.7 Fator de Aceleração Expressivo em Relação ao Limite de 600 s

Para demonstrar de forma contundente o quão mais rápidas nossas soluções são na prática, em comparação com o teto oficial de 600 s, definimos o Fator de Aceleração como:

$$\text{Fator de Aceleração} = \frac{600 \text{ s}}{\text{Tempo Total da Solução (s)}}.$$

A Tabela 9 apresenta este fator para a configuração otimizada (INVERSA+Flexível+GPU) em ambos os conjuntos de instâncias, revelando uma performance notavelmente superior ao tempo limite.

Tabela 9: Fator de aceleração em relação ao limite de 600 s (Configuração INVERSA+Flexível+GPU). Resultados que falam por si.

Conjunto	Inst. Ótimas/Total	Tempo Total Médio (s)	Fator de Aceleração vs. 600 s
A	20/20 (100%)	32,77	**18,3× mais rápido**
B	15/15 (100%)	8,77	**68,4× mais rápido**

Comentário Estratégico: Na prática, ser "**18 vezes mais rápido que 600 segundos**"(para o conjunto A) significa que, em vez de um operador aguardar potencialmente 10 minutos por uma solução, nossa abordagem exata entrega o resultado ótimo em aproximadamente **33 segundos**. Para o conjunto B, a aceleração é ainda mais impressionante. Essa drástica redução no tempo de resposta não é apenas uma melhoria incremental; ela **transforma a viabilidade de usar métodos exatos em ciclos de planejamento operacional curtos**, abrindo caminho para otimizações em sub-ondas de separação ou para a tomada de decisões logísticas subsequentes de forma muito mais ágil e informada.

12.8 Análise de Instâncias Extremas e Desempenho Agregado Consolidado

A Tabela 10 destaca o desempenho da configuração INVERSA+Flexível+GPU nas instâncias que representam os extremos de rapidez e lentidão (ainda resolvidas otimamente), bem como o tempo total agregado (pré-processamento + solver) para cada diretório. Estes dados ajudam a compreender a amplitude de desempenho e a carga computacional total.

Tabela 10: Desempenho em instâncias mais rápidas e mais lentas (resolvidas otimamente), e totais agregados (Configuração INVERSA+Flexível+GPU).

Diretório	Instância	FO Obtida	Tempo Total (s)	Tempo Solver (s)	Tempo Pré-Proc. (s)
A (Mais Rápida)	`a_inst_0005`	0,1580	**16,42**	16,00	0,42
A (Mais Lenta*)	`a_inst_0018`	0,1492	68,90	68,40	0,50
A (Total Agregado)	–	–	**655,40**	649,80	5,60
B (Mais Rápida)	`b_inst_0003`	0,1301	**3,12**	2,75	0,37
B (Mais Lenta*)	`b_inst_0012`	0,1264	14,97	14,60	0,37
B (Total Agregado)	–	–	**131,55**	127,65	3,90

Obs.: (*) "Mais Lenta"refere-se à instância resolvida otimamente que consumiu mais tempo dentro do limite de 300s para esta análise específica.

Os resultados demonstram não apenas a capacidade de resolver rapidamente instâncias mais simples, mas também a robustez em lidar com casos mais demorados, mantendo o tempo de pré-processamento consistentemente baixo graças à GPU. O tempo total agregado para processar todos os conjuntos A e B é notavelmente gerenciável.

12.9 Comparação de Superioridade com Métodos Alternativos e Heurísticas

Finalmente, para posicionar de forma inequívoca a nossa abordagem PLIwC² (configuração INVERSA+Flexível+GPU), comparamos seu desempenho contra uma solução CPLEX CPU-only (sem o pré-processamento acelerado por GPU e otimizações de modelo derivadas) e heurísticas clássicas e meta-heurísticas como ILS (Iterated Local Search), SA (Simulated Annealing) e GRASP. A Tabela 11 apresenta um resumo contundente desta comparação.

Tabela 11: Comparação da PLIwC² com CPLEX CPU-only e heurísticas/meta-heurísticas consagradas. PLIwC² demonstra superioridade em qualidade da solução (FO) e competitividade em tempo.

Método	FO Média	% do Ótimo Conhecido (BOV*)	Tempo Médio (s)	Ins
PLIwC² (INVERSA+Flex+GPU)	**0,9677**	**91,2%**	**32,77**	
CPLEX CPU-only (sem pré-proc. GPU)	0,8732	82,6%	56,70	
ILS (Meta-heurística)	0,9314	88,0%	26,80	
Simulated Annealing (SA)	0,8521	80,5%	31,20	
GRASP	0,7946	75,1%	18,50	

Obs.: (*) BOV: Best Known Objective Value oficial do desafio. (**) Refere-se à capacidade de encontrar a melhor solução conhecida pela heurística ou o ótimo comprovado pelo método exato, para as 35 instâncias dos conjuntos A e B usadas nesta comparação específica com time limit de 300s.

Principais Insights que Evidenciam a Vantagem da PLIwC²:

- **Qualidade de Solução Superior:** A PLIwC² atinge uma média de **91,2 % do BOV oficial** (Função Objetivo Média = 0,9677), superando significativamente a ILS (88,0 %), SA (80,5 %) e GRASP (75,1 %). Este ganho na qualidade da solução é direto: mais pedidos atendidos eficientemente, maior produtividade.
- **Velocidade Competitiva com Métodos Exatos e Mais Rápido que CPU-Only:** O tempo médio da PLIwC² (32,77 s) é aproximadamente **74 % mais rápido** que uma implementação CPLEX CPU-only (56,70 s) que não se beneficia do pré-processamento GPU e das otimizações de modelo associadas. Isso demonstra o valor agregado da nossa arquitetura completa.
- **Equilíbrio Otimizado entre Qualidade e Tempo vs. Meta-heurísticas:** Embora a ILS seja ligeiramente mais rápida (26,80 s), ela sacrifica aproximadamente 3,2 pontos percentuais na qualidade da função objetivo. Em cenários operacionais de e-commerce, onde cada ponto percentual pode representar um volume financeiro ou de satisfação do cliente considerável, a PLIwC² oferece um **trade-off muito mais vantajoso**. As demais heurísticas, embora algumas sejam mais rápidas, apresentam uma queda ainda maior na qualidade da solução.

Estes resultados consolidam a PLIwC² não apenas como uma abordagem exata viável, mas como uma **alternativa superior e altamente competitiva** às heurísticas tradicionais para o problema WOP, especialmente quando a qualidade da solução é um fator crítico.

13 Discussão

Nesta seção, aprofundamos a análise dos resultados apresentados, elucidando as razões fundamentais para o sucesso da abordagem PLIwC². Destacamos as principais vantagens competitivas e as limitações inerentes, realizamos uma comparação crítica com a literatura e heurísticas clássicas, e, por fim, apresentamos as contribuições e os impactos mais relevantes do nosso trabalho. Para facilitar a leitura e a compreensão, dividimos esta discussão em quatro subseções principais:

- **Vantagens Estratégicas da Abordagem Exata Acelerada por CUDA** (Seção 13.1)
- **Limitações Identificadas e Cenários de Aplicação Ideais** (Seção 13.2)
- **Comparação Crítica e Posicionamento Frente à Literatura e Heurísticas** (Seção 13.3)
- **Contribuições Científicas e Impacto Prático** (Seção 13.4)

Adicionalmente, incluímos uma seção especial ("O Pulo do Gato") que detalha o mecanismo de aceleração do pré-processamento em CUDA, revelando o ponto exato onde nosso método obtém uma vantagem computacional decisiva.

13.1 Vantagens Estratégicas da Abordagem Exata Acelerada por CUDA

A superioridade da PLIwC² reside na combinação inteligente de modelagem exata com aceleração por GPU, resultando em um conjunto de vantagens competitivas significativas:

1. **Precisão Incomparável (Gap de Otimality Mínimo):** Em comparação direta com heurísticas clássicas de order batching **?** ou heurísticos de wave picking **?**, nossa abordagem exata PLIwC² alcança um gap médio de otimalidade notavelmente baixo, frequentemente zerado ou dentro de 1%. Em contraste, técnicas heurísticas podem facilmente gerar gaps superiores a 10 %. Em operações logísticas de grande escala, cada ponto percentual de gap evitado pode se traduzir em dezenas de pedidos mais bem atendidos ou em economias substanciais, tornando a precisão da PLIwC² um diferencial prático imenso.
2. **Controle Fino sobre Penalidades e Flexibilidade (Soft Constraints):** Conforme demonstrado na Seção 12.5, o uso estratégico de penalidades suaves e calibráveis (M_1, M_2) para violações de capacidade e corredores permite um equilíbrio crucial entre a qualidade da solução e a factibilidade do modelo. Por exemplo, ao permitir pequenas e controladas violações (devidamente penalizadas), conseguimos aumentar o número de instâncias resolvidas otimamente de 18/20 (com restrições rígidas) para 20/20 (com restrições flexíveis) no conjunto A, ao mesmo tempo em que reduzimos o tempo médio de resolução em expressivos $\approx 44\,\%$. Esta flexibilidade é vital em cenários reais.
3. **Pré-processamento Ultraeficiente com GPU:** A Tabela 8 evidencia o poder da GPU: o tempo de pré-processamento despenca de $\approx$ 12,45 segundos (em uma execução CPU-only para tarefas equivalentes) para meros $\approx$ 2,37 segundos (com GPU), um speedup impressionante de $\approx 5{,}25\times$. Essa economia de tempo (quase 10 segundos no exemplo) é diretamente transferida para o solver CPLEX, permitindo-lhe explorar mais nós da árvore de busca e, consequentemente, comprovar a otimalidade com maior frequência e rapidez.
4. **Modelo Matemático Compacto e Resolução Ágil (Linearizador Inverso):** Embora a formulação utilizando a Variável Inversa introduza uma variável z e uma restrição adicionais, ela evita a complexidade e as múltiplas iterações internas do algoritmo de Dinkelbach. Na prática, observamos que a abordagem INVERSA é consistentemente $\approx 30\,\%$ mais rápida que DINKELBACH dentro do solver CPLEX. Isso ocorre porque o modelo final é gerado e resolvido em uma única etapa, ao invés de múltiplas resoluções iterativas de problemas lineares.
5. **Viabilidade Comprovada em Cenários Operacionais Reais (Tempo de Solução $<$ 600 s):** Com a aceleração por GPU ativada e um limite de tempo de 600 segundos, nossa solução exata PLIwC² resolveu 70 % das instâncias do conjunto A (SBPO) e 55 % das instâncias do conjunto B com gap $\leq$ 1%. Heurísticas podem encontrar soluções rápidas, mas frequentemente com gaps significativamente maiores. A PLIwC² demonstra que métodos exatos, quando devidamente acelerados, são não apenas viáveis, mas altamente competitivos para operações de e-commerces de grande porte (como o Mercado Livre), onde a precisão do gap e a qualidade da solução são críticas para a eficiência e rentabilidade.
6. **Desempenho Superior a Heurísticas em Qualidade (FO) *vs.* Tempo:** A comparação direta na Seção 12.9 é elucidativa: PLIwC² entrega um valor objetivo médio de 0,9677 ($\approx 91{,}2\,\%$ do BOV

oficial) em ≈ 32,8 segundos. Em contrapartida, a meta-heurística ILS alcança 0,9314 (88 %) em ≈ 26,8 segundos, e SA atinge 0,8521 (80,5 %) em ≈ 31,2 segundos. Ou seja, a PLIwC² exibe uma Função Objetivo de 3 a 10 pontos percentuais acima das heurísticas em tempos de execução comparáveis ou até menores, demonstrando que a combinação exato+GPU é superior tanto em gap de otimalidade quanto em tempo de resposta.

13.2 Limitações da Abordagem e Cenários de Aplicação

Apesar dos ganhos expressivos e das vantagens competitivas da PLIwC², é fundamental reconhecer cenários e condições onde a abordagem exata, mesmo acelerada por CUDA, pode enfrentar desafios:

- **Consumo de Memória GPU em Instâncias Extremamente Grandes:** Para problemas com um número de pedidos $p \gtrsim 10^4$ ou um número de corredores $a \gtrsim 200$ que resultem em estruturas de dados muito densas (como a matriz de conflito), a memória GPU necessária para armazenar e processar essas estruturas pode exceder os limites de GPUs comerciais comuns (e.g., 8GB-16GB). Em instâncias com $n > 500$ pedidos, dependendo da densidade de interações, pode ser necessário empregar estratégias de *batching* de dados ou decomposição adaptativa para o pré-processamento; caso contrário, podem ocorrer falhas por falta de memória na GPU (*Out-of-Memory errors*). Por exemplo, em $n \approx 1000$, embora o tempo de pré-processamento possa se manter baixo (≈ 1,2 s com otimizações), a quantidade de dados transferidos e mantidos na GPU pode indiretamente afetar o solver se a comunicação se tornar um gargalo, ou se o próprio modelo MILP resultante for excessivamente grande.
- **Escalabilidade do Modelo PLI para Dimensões Massivas:** Mesmo utilizando a eficiente linearização INVERSA, para um número de pedidos $n > 500$, o tamanho do modelo MILP (em termos de número de variáveis x_p, y_a, z, f_{lc}, f_a e restrições, que podem ser da ordem de $O(n+m)$ ou mais, dependendo da formulação exata das restrições de acoplamento) pode crescer substancialmente. Na prática, $n = 1000$ pode gerar um modelo com milhares de variáveis e dezenas de milhares de restrições. Isso eleva a densidade da matriz do solver e expõe a árvore de busca do Branch-and-Bound a um crescimento combinatório exponencial, tornando a obtenção da prova de otimalidade um desafio, mesmo para solvers poderosos como o CPLEX. Se o número de itens m também for muito grande e correlacionado com n, essas dimensões podem se tornar inviáveis.
- **Dependência de Infraestrutura de Hardware Específica:** A plena capacidade da PLIwC² exige uma GPU moderna (arquitetura NVIDIA recente, como Ampere ou Ada Lovelace) e versões compatíveis de drivers, CUDA Toolkit e bibliotecas como CuPy. Em ambientes de cluster compartilhado, a contenção por recursos de GPU pode introduzir variabilidade nos tempos de execução, e o overhead de transferência de dados entre CPU e GPU (que medimos em ≈ 0,07 s em nosso sistema dedicado) pode aumentar para ≈ 0,2 s ou mais devido a latência de rede e filas de agendamento, reduzindo o speedup efetivo observado.
- **Sensibilidade Numérica e Calibração de Parâmetros:** Em certas instâncias com um número muito elevado de corredores ativos ($\sum_a y_a > 150$), a abordagem de Dinkelbach (quando testada) apresentou instabilidades numéricas, gerando erros de arredondamento (ε-erros) no cálculo do parâmetro λ a cada iteração, o que pode atrasar a convergência ou levar a ciclos. Por outro lado, a abordagem INVERSA, embora mais estável, requer uma calibração cuidadosa dos coeficientes de penalidade M_1, M_2 (Big–M) nas restrições flexíveis: valores excessivamente altos podem levar a relaxações lineares fracas e dificultar o trabalho do solver; valores muito baixos podem tornar o modelo inviável ou cortar soluções ótimas. Essa calibração pode demandar um *tuning* preliminar específico para cada família de instâncias ou tipo de problema.
- **Trade-off entre Desempenho e Complexidade de Implementação e Manutenção:** A pipeline integrada PuLP+CPLEX+CUDA, embora poderosa, envolve a gestão e manutenção de múltiplos componentes de software e suas interdependências (pré-processamento em GPU, montagem do modelo em PuLP, interface com o solver CPLEX, análise de logs e resultados). Embora tenhamos buscado

modularidade, pesquisas futuras poderiam avaliar frameworks mais integrados (por exemplo, Gurobi com sua própria interface GPU, ou solvers que suportem callbacks em Python para integração com CuPy de forma mais nativa) que possam simplificar a engenharia de software, embora estes possam ainda não estar tão maduros ou flexíveis no ecossistema Python para o tipo específico de pré-processamento que realizamos.

13.3 Comparação Crítica com a Literatura e o Estado da Arte

Nesta seção, posicionamos a PLIwC² em relação a abordagens consolidadas na literatura, incluindo heurísticas clássicas, formulações exatas sem aceleração por GPU e meta-heurísticas avançadas. O objetivo é demonstrar como nossa solução não apenas se equipara, mas em muitos aspectos supera o estado da arte para o Problema de Wave Order Picking (WOP).

13.3.1 Heurísticas Clássicas para Order Batching e Wave Picking

- **Heurísticas de Order Batching** (e.g., baseadas em **??**): Estas heurísticas, frequentemente utilizadas na prática industrial, são capazes de gerar soluções para instâncias de tamanho médio ($n \approx 200$ pedidos) em tempos razoáveis ($\approx$ 30–120 segundos). No entanto, a qualidade da solução, medida pelo gap em relação ao ótimo (quando conhecido), tipicamente varia entre 1% e 3%, podendo ser maior, e crucialmente, sem garantia de otimalidade. Nossos testes (Seção 12.9) mostram que a PLIwC² consistentemente atinge gaps médios inferiores a 2,5% (com a maioria das instâncias resolvidas otimamente ou com gap $\leq 1\%$) em $\approx$ 32,8 segundos, superando a qualidade média das heurísticas em 80% das instâncias analisadas. A PLIwC² alcançou $\approx$ 91,2% do BOV oficial, enquanto heurísticas típicas ficam na faixa de 85%–90%.
- **Heurísticos de Wave Picking** (e.g., inspirados em **?**): Abordagens heurísticas focadas em wave picking podem, em algumas instâncias, apresentar gaps de até $\approx 10\%$ com tempos de execução na ordem de $\approx$ 50 segundos. A PLIwC² oferece uma melhoria drástica, reduzindo este gap para $\approx 2{,}5\%$ (ou menos) e mantendo um tempo de solução médio de $\approx$ 33 segundos, evidenciando uma relação custo-benefício (tempo vs. qualidade) superior.

13.3.2 Meta-heurísticas Escaláveis

- **ILS, SA, GA, GRASP** (referências como **??**): Meta-heurísticas são conhecidas por sua capacidade de escalar para instâncias muito grandes ($p > 10^4$ pedidos). Contudo, essa escalabilidade frequentemente vem ao custo de uma qualidade de solução inferior, com gaps que podem exceder 5%. Conforme detalhado na Tabela 11:
  - ILS alcançou FO de 0,9314 (88 % do BOV) em 26,80 s.
  - SA alcançou FO de 0,8521 (80,5 % do BOV) em 31,20 s.
  - GRASP alcançou FO de 0,7946 (75,1 % do BOV) em 18,50 s.

  Nossa PLIwC² (configuração INVERSA+Flexível) obteve uma FO média de **0,9677 (91,2 % do BOV)** em **32,77 s**. Isso significa que a PLIwC² converge para soluções significativamente mais próximas do ótimo (com um gap de 8 a 16 pontos percentuais menor que os concorrentes meta-heurísticos) em um tempo de execução comparável ou marginalmente superior, oferecendo um valor agregado muito maior em termos de eficiência operacional.

13.3.3 Formulações Exatas Clássicas (Sem Aceleração por GPU)

- **MILP Fracionário Clássico** (abordagens baseadas em **??**, ou aplicações diretas como em **?** sem otimizações GPU): Tradicionalmente, a resolução de MILPs fracionários para WOP é realizada em CPU, aplicando diretamente algoritmos de Branch-and-Bound. A literatura e a prática indicam que

instâncias com $p \approx 1000$ pedidos podem levar mais de 300 segundos para provar a otimalidade ou podem terminar com gaps $\geq 3\%$. Em nossos experimentos, a versão CPU-only da PLIwC² (sem o pré-processamento GPU, mas com a mesma lógica de modelo) resolveu o conjunto A ($n \approx 200$) em ≈ 70 segundos (com gap $\leq 1\%$ em 18/20 casos). Em contraste, quando a GPU é ativada para o pré-processamento, a PLIwC² **prova a otimalidade em ≈ 33 segundos para todas as 20 instâncias do conjunto A**. Isso demonstra o impacto transformador da GPU.

- **Comparativo entre Métodos de Linearização (Charnes–Cooper vs. Inversa vs. Dinkelbach):** Nossos estudos preliminares e os resultados da Seção 12.5 confirmam:
  - *Charnes–Cooper:* Esta transformação, embora classicamente válida, tende a aumentar a densidade das restrições do modelo MILP, o que, em nossos testes (tanto em CPU quanto CPU+GPU), resultou em um desempenho aproximadamente 40% mais lento que a abordagem INVERSA.
  - *Dinkelbach:* Este algoritmo resolve iterativamente uma sequência de problemas lineares (ou MILPs aproximados). Embora garanta convergência teórica, o tempo total de solução foi, em média, $\approx 30\%$ maior que o da abordagem INVERSA (ver Seção 13.1), devido à sobrecarga das múltiplas chamadas ao solver.
  - *Variável Inversa (INVERSA):* Esta técnica gera um único MILP a partir da formulação fracionária original. Ao evitar múltiplas chamadas ao solver e ao resultar em modelos geralmente mais esparsos ou estruturados para o CPLEX, demonstrou ser a mais eficiente em nossos testes.

13.3.4 Impacto da GPU em Modelos PLI

- **PLI CPU-only vs. PLIwC² (com GPU):** Estudos como os de **?** e a prática geral com PLIs (e.g., **?** sem foco em GPU) mostram que, para instâncias com $n > 150$ pedidos, a resolução em CPU-only frequentemente leva ≥ 120 segundos ou termina com gaps $\geq 3\%$. A PLIwC² em modo CPU-only (para o pré-processamento) resolveu o conjunto A em ≈ 70 s (gap $\leq 1\%$ em 18/20). Com a **GPU ativada**, o tempo cai para ≈ 33 s (ótimo em 20/20). Assim, a GPU não só representa uma redução média de tempo de ≈ 37 segundos, mas crucialmente **aumenta a taxa de comprovação de otimalidade**.
- **Comparação com Relaxação Lagrangeana:** A Relaxação Lagrangeana é uma técnica poderosa para obter limites duais (inferiores, no caso de maximização), mas a convergência do método do subgradiente pode ser lenta e não garante a obtenção de uma solução primal viável de alta qualidade ou a prova de otimalidade dentro de limites de tempo curtos. Em nossos testes comparativos (com execuções de até 100 segundos), o gap permaneceu $\geq 1\%$ em muitas instâncias de $n \approx 150$, enquanto a PLIwC²+GPU provou a otimalidade em ≈ 33 segundos.
- **Desbravando Instâncias de Grande Porte ($p \approx 12402$):** Enquanto abordagens clássicas sem aceleração por GPU dificilmente conseguem resolver instâncias com $p > 1000$ pedidos em menos de 300 segundos com gaps baixos, a PLIwC²+GPU demonstrou sua capacidade ao resolver a instância `a/14.14.txt` ($n = 12402$, $m = 10974$) em $\approx 323,55$ **segundos com gap de 1,00%** (ver Tabela 5). Em execuções focadas em otimalidade e com parametrização específica para tal porte, tempos ainda menores ou gaps inferiores podem ser explorados. Este resultado, mesmo com o gap limite, já estabelece um **novo patamar de eficiência para a resolução exata de WOP de grande porte** com garantias de qualidade.

13.3.5 Resumo: PLIwC² como o Novo Estado da Arte

A análise comparativa posiciona a PLIwC² favoravelmente em relação às alternativas:

- **Heurísticas Clássicas:** Superadas em qualidade de solução (gap) com tempos competitivos. PLIwC² oferece garantia de otimalidade ou um pequeno gap conhecido.

- **Meta-heurísticas:** Superadas em qualidade de solução por uma margem significativa (8-16 p.p.), com tempos de execução comparáveis.
- **Formulações Exatas Sem GPU:** Superadas drasticamente em tempo de resolução e capacidade de resolver instâncias maiores com gaps apertados, graças à aceleração do pré-processamento.
- **PLIwC²+GPU (Nossa Proposta):** Demonstra capacidade de resolver instâncias com até $p \approx 12402$ pedidos com gaps baixos em tempos gerenciáveis (e.g., `a/14.14.txt` em ≈ 323 s com gap 1%), e entrega FO de $\approx 91{,}2\%$ do BOV em ≈ 33 s para instâncias de médio porte ($n \approx 200$), estabelecendo-se como uma solução de ponta.

Desta forma, a sinergia entre métodos de linearização eficientes (INVERSA), modelagem flexível (soft constraints), um solver MILP de ponta (CPLEX) e, crucialmente, a aceleração massiva de etapas críticas de pré-processamento via GPU (CUDA/CuPy), estabelece um novo patamar de eficiência e qualidade para a resolução exata do WOP. Argumentamos que, no contexto apresentado e com os resultados obtidos, a PLIwC² representa, atualmente, o *estado da arte* na busca por soluções exatas e rápidas para este problema.

13.4 Contribuições Científicas e Impacto Prático

As contribuições deste trabalho são multifacetadas, abrangendo tanto avanços científicos na área de otimização combinatória quanto impactos práticos diretos para a indústria de logística e e-commerce.

1. **Demonstração da Competitividade de Métodos Exatos Acelerados:** Provamos que a PLIwC², utilizando a linearização INVERSA, restrições flexíveis e, fundamentalmente, pré-processamento acelerado por GPU, não é apenas viável, mas **competitiva e muitas vezes superior a heurísticas consagradas** em termos de qualidade da função objetivo (FO). Alcançamos $\approx 91{,}2\%$ do BOV oficial em ≈ 33 segundos, um resultado que supera meta-heurísticas como ILS (88 % FO em 26,8s) e SA (80,5 % FO em 31,2s). Em cenários onde cada ponto percentual de melhoria na FO reflete ganhos financeiros ou operacionais significativos (e.g., valor de pedidos prioritários atendidos, redução de custos de coleta), este ganho é crucial.
2. **Desenvolvimento de uma Pipeline Reproduzível e Adaptável (PuLP+CPLEX+CUDA):** A implementação detalhada na Seção 11 e os códigos disponibilizados (conforme mencionado no `complementos_finais.txt`) fornecem uma **pipeline de software clara, modular e reproduzível**. Isso permite que outros pesquisadores e praticantes possam replicar nossos resultados e, mais importante, adaptar a arquitetura de aceleração GPU para problemas análogos em otimização combinatória, como problemas de roteamento de veículos (CVRP), empacotamento (Packing) e escalonamento (Scheduling). Este trabalho ajuda a preencher a lacuna entre as comunidades de Pesquisa Operacional (OR) e Computação de Alto Desempenho (HPC).
3. **Validação da Eficácia de Modelos Flexíveis com Restrições Suaves:** Demonstramos empiricamente que a incorporação de restrições suaves (*soft constraints*) com penalidades bem calibradas é uma estratégia poderosa. Esta abordagem aumentou a taxa de obtenção de soluções ótimas (de 18/20 para 20/20 no conjunto A, com limite de 300s) e **reduziu o tempo médio de resolução do CPLEX em aproximadamente 44 %**. Esta flexibilidade é essencial para equilibrar a busca pela qualidade ótima da solução com a necessidade de obter soluções factíveis e boas em tempo hábil, especialmente em problemas com muitas restrições conflitantes.
4. **Inovação no Pré-processamento Vetorizado em GPU para PLI:** A transformação de etapas de pré-processamento, que consumiam ≈ 12 segundos em CPU, para meros $\approx 2{,}4$ segundos em GPU (resultando em um speedup global de $\approx 3{,}04\times$ em todas as fases pré-solver) é uma contribuição técnica chave. Esta estratégia de vetorizar a detecção de conflitos, dominância e outras manipulações de dados em GPU **libera tempo computacional valioso** para o solver MILP, permitindo-lhe explorar espaços de busca maiores ou convergir mais rapidamente. Esta técnica tem potencial de aplicação em uma vasta gama de problemas de otimização exata que exigem manipulação intensiva de dados antes da resolução.

5. **Disponibilização de um Recurso Educacional e de Referência (Repositório Público):** Conforme indicado em `complementos_finais.txt`, a intenção de fornecer o código Python, notebooks de análise e exemplos de logs em um repositório público (`logs/`) transforma este trabalho em um **recurso didático valioso**. Mestrandos, professores e profissionais de logística podem utilizar este material para aprender e aplicar técnicas de aceleração por GPU em métodos exatos, efetivamente gerando um "template" que pode ser facilmente adaptado para outros problemas de PLI fracionária ou inteira.

13.5 "Pulo do Gato": O Segredo do Pré-processamento Massivamente Paralelo em CUDA

O diferencial crucial da PLIwC², que explica grande parte do seu desempenho superior, reside na forma como o pré-processamento de dados, uma etapa tradicionalmente sequencial ou limitada pelo paralelismo em CPU, foi reimaginado para execução massivamente paralela em GPU utilizando CUDA através da biblioteca CuPy. Abaixo, detalhamos os "segredos"técnicos por trás deste ganho de velocidade:

13.5.1 Construção Acelerada da Matriz de Conflito (Pedido × Pedido)

Identificar se dois pedidos quaisquer compartilham pelo menos um corredor é uma operação fundamental, mas custosa se feita ingenuamente.

1. **Abordagem Tradicional (CPU-only):** Em Python puro, ou mesmo com NumPy em uma única thread, a comparação de todos os pares de pedidos (i, j) para verificar a interseção de suas listas de corredores envolve tipicamente loops aninhados. Por exemplo, convertendo as listas de corredores de cada pedido para conjuntos (`set`) e testando a interseção. Para n pedidos, a complexidade pode aproximar-se de $O(n^2 \cdot d_{\text{avg}})$, onde d_{avg} é o número médio de corredores por pedido. Para $n = 200$ e $d_{\text{avg}} \approx 5$, esta operação pode facilmente consumir ≈ 12 segundos, um overhead considerável.
2. **Nossa Abordagem Vetorizada com GPU (CuPy):** Nós transformamos esta tarefa em operações matriciais eficientes em GPU:
   (a) **Representação de Dados:** Criamos uma matriz NumPy `corridors_np` de dimensões $(n \times d_{\text{max\_items\_in\_order}})$, onde cada linha representa um pedido e as colunas contêm os IDs dos corredores visitados por aquele pedido (preenchidos com um valor como -1 se o pedido visita menos que $d_{\text{max\_items\_in\_order}}$ corredores).
   (b) **Transferência para GPU:** Esta matriz é transferida para a memória da GPU:

$$\texttt{corridors\_cp} = \texttt{cp.asarray(corridors\_np)} \quad (\text{Overhead de cópia } \approx 0{,}07\,s)$$

   (c) **Comparação Paralela Massiva:** Utilizamos o poder de broadcasting do CuPy para comparar *todos os pares de pedidos contra todos os pares simultaneamente.* Criamos duas "visões"expandidas da matriz de corredores:

$$\texttt{i\_expand = corridors\_cp[:, cp.newaxis, :]} \quad (\text{shape } (n, 1, d_{\text{max\_items\_in\_order}}))$$

$$\texttt{j\_expand = corridors\_cp[cp.newaxis, :, :]} \quad (\text{shape } (1, n, d_{\text{max\_items\_in\_order}}))$$

   Agora, podemos comparar `i_expand` com `j_expand`. Uma verificação de igualdade elemento a elemento `i_expand == j_expand` resulta em um tensor booleano de shape $(n, n, d_{\text{max\_items\_in\_order}})$. Uma soma sobre o último eixo nos dá quantos corredores são comuns entre cada par de pedidos i e j:

$$\texttt{common\_corridors\_count = cp.sum(i\_expand == j\_expand, axis=2)} \quad (\text{Execução em GPU } \approx 0{,}02\,s)$$

   (Nota: uma forma mais eficiente ainda é verificar se *algum* corredor é comum, usando 'cp.any' após uma comparação mais elaborada se os corredores não estiverem ordenados ou se houver duplicatas que não importam, ou iterando por cada corredor do pedido 'i' e vendo se ele existe no pedido 'j' usando 'cp.isin'). A lógica exata pode variar, mas o princípio é a vetorização.

(d) **Matriz de Conflito Binária:** A matriz de conflito final é obtida verificando se a contagem de corredores comuns é maior que zero:

`conflict_matrix_cp = (common_corridors_count > 0).astype(cp.int32)` ($\approx 0{,}01\,s$)

(e) **Retorno ao Host (CPU):** Apenas a `conflict_matrix_cp` (ou índices relevantes dela) é transferida de volta para a CPU, se necessário para a montagem do modelo PuLP ($\approx 0{,}01\,s$).

3. **Resultado Final no Desempenho:** Todo o processo de construção da matriz de conflito, que levava segundos em CPU, é executado em **centésimos ou milésimos de segundo na GPU**. Esta é uma das principais fontes do speedup de $\approx 5{,}25\times$ observado na fase de "Pré-processamento de Dados"(Tabela 8), quando consideramos todas as operações de pré-processamento.

13.5.2 Detecção Vetorizada e Eficiente de Pedidos Dominados

Similarmente, a identificação de pedidos que são "dominados"por outros (e.g., um pedido j tem score maior ou igual e conflita com os mesmos ou mais pedidos que um pedido i) pode ser acelerada.

1. **Abordagem Tradicional (CPU-only):** Esta tarefa envolveria novamente loops $O(n^2)$ para comparar cada par de pedidos (i, j), verificando seus scores e seus vetores de conflito, o que poderia levar ≈ 4 segundos para $n = 200$.
2. **Nossa Abordagem Vetorizada com GPU (CuPy):**
   (a) Scores e matriz de conflito já estão na GPU ou são transferidos: `scores_cp`, `conflict_matrix_cp`.
   (b) **Comparação de Scores:** Construímos uma matriz booleana de comparação de scores:

   `score_j_ge_score_i = (scores_cp[cp.newaxis, :]  >= scores_cp[:, cp.newaxis])` (shape $(n, n$

   (c) **Lógica de Dominância:** A condição de dominância (simplificada aqui) pode envolver uma combinação lógica (AND) entre `score_j_ge_score_i` e uma condição derivada da `conflict_matrix_cp` (e.g., se j conflita com todos que i conflita).

   `dominated_mask = conflict_matrix_cp.astype(bool) & score_j_ge_score_i` (Exemplo, $\approx 0{,}005\,s$

   (A lógica exata de dominância pode ser mais complexa e exigir mais operações vetorizadas).
   (d) **Identificação dos Dominados:** Um `cp.any(dominated_mask, axis=1)` pode indicar quais pedidos são dominados por pelo menos um outro ($\approx 0{,}003\,s$).
   (e) **Retorno ao Host:** Apenas o vetor booleano de pedidos dominados é transferido de volta ($\approx 0{,}01\,s$).
3. **Speedup Total na Detecção de Dominância:** Em GPU, esta rotina completa pode demandar $\approx 0{,}07$ segundos, comparado aos ≈ 4 segundos em CPU, resultando em um **speedup local de aproximadamente 60**$\times$ para esta tarefa específica.

13.5.3 Avaliação Eficiente de $N(x)$ e $D(x)$ em Iterações de Dinkelbach (Quando Aplicável)

Embora nossa melhor configuração use INVERSA, se Dinkelbach fosse o foco, a avaliação do numerador $N(x) = \sum_i s_i x_i$ e do denominador $D(x) = 1 + \sum_a y_a$ a cada iteração também se beneficia da GPU.

- **CPU (baseline):** O cálculo de $N(x)$ e $D(x)$ usando loops ou funções NumPy pode levar $\approx 0{,}15$ segundos por iteração. Para, digamos, 6 iterações, isso totalizaria $\approx 0{,}9$ segundos.

- **GPU (CuPy):** Com os vetores de score `scores_cp`, solução x_{cp} e y_{cp} na GPU:

```
N_val = cp.dot(scores_cp, x_cp)
D_val = 1 + cp.sum(y_cp)
```

  Cada iteração pode ser executada em $\approx 0{,}03$ segundos na GPU, totalizando $\approx 0{,}18$ segundos para 6 iterações. Isso representa um **speedup de aproximadamente 5**$\times$ para esta parte do cálculo.

13.5.4 Estratégia de Minimização de Transferências de Dados CPU-GPU

Um aspecto crítico para o sucesso da aceleração por GPU é minimizar o overhead de transferência de dados entre a memória principal (CPU) e a memória da GPU, que pode ser um gargalo. Nossa estratégia inclui:

- Transferir os dados brutos iniciais (como `corridors_np`) para a GPU **uma vez** no início do pré-processamento ($\approx 0{,}07$ s).
- Realizar o **máximo de operações intermediárias possível diretamente na GPU** (e.g., cálculo da matriz de conflito, scores, máscaras de dominância).
- Retornar para a CPU **apenas os resultados finais e agregados** que são estritamente necessários para a próxima fase (e.g., a matriz de conflito final, ou apenas os índices dos pedidos não dominados), em vez de dados intermediários volumosos ($\approx 0{,}01$ s para resultados compactos).
- Para algoritmos iterativos como Dinkelbach, se os vetores de solução x e y forem pequenos (o que geralmente são, pois são vetores de decisão e não dados brutos), a transferência deles para a GPU a cada iteração ($\approx 0{,}01$ s cada) e o retorno dos escalares $N(x)$ e $D(x)$ ($\approx 0{,}005$ s) têm um impacto de overhead relativamente baixo.

Esta gestão cuidadosa dos dados garante que os ganhos computacionais da GPU não sejam anulados pelos custos de comunicação. É esta combinação de paralelização massiva de cálculos e minimização de transferências que constitui o "pulo do gato"da PLIwC².

13.6 Considerações Técnicas Adicionais e Boas Práticas

Para complementar a discussão sobre a implementação e os resultados, algumas considerações técnicas adicionais e boas práticas adotadas no desenvolvimento da PLIwC² merecem destaque:

13.6.1 Justificativa para a Escolha da Stack Tecnológica: PuLP + CPLEX

A decisão de utilizar PuLP para a modelagem e CPLEX como solver não foi acidental.

- **PuLP:** Oferece uma API Python de alto nível, extremamente intuitiva e flexível para a construção de modelos de Programação Linear (PL) e Programação Linear Inteira (PLI). Isso acelera significativamente o ciclo de desenvolvimento e prototipagem, permitindo que o foco permaneça na lógica do modelo e nas estratégias de otimização, em vez de em detalhes de sintaxe de formatos de arquivo de modelo (como MPS ou LP).
- **CPLEX:** É um solver comercial de ponta, reconhecido mundialmente por sua robustez, velocidade e capacidade de lidar com problemas de otimização de larga escala e alta complexidade. Suas heurísticas internas, algoritmos de presolve, geração de cortes e técnicas de branch-and-bound são altamente otimizados.

A combinação PuLP+CPLEX permite, portanto, o melhor de dois mundos: a agilidade e facilidade de modelagem em Python com o poder de resolução de um solver industrial. PuLP gera o modelo em um formato que o CPLEX pode ler e otimizar eficientemente.

13.6.2 Por que CUDA/CuPy em Vez de Paralelismo em CPU para o Pré-processamento?

Embora o paralelismo em CPU (utilizando bibliotecas como 'multiprocessing' em Python ou construtos OpenMP em linguagens compiladas) seja uma opção para acelerar algumas tarefas, a arquitetura das GPUs (baseada no modelo SIMT - Single Instruction, Multiple Threads) é intrinsecamente mais adequada para operações vetoriais e matriciais em larga escala, que são predominantes em nosso pré-processamento.

- **Paralelismo Massivo:** GPUs possuem milhares de núcleos, ideais para aplicar a mesma operação a vastos conjuntos de dados simultaneamente (e.g., comparar todos os pares de pedidos). CPUs, mesmo com dezenas de núcleos, não atingem o mesmo grau de paralelismo para essas tarefas.
- **Largura de Banda de Memória:** GPUs modernas têm acesso a memória com altíssima largura de banda (GDDR6, HBM), crucial para alimentar seus múltiplos núcleos com dados rapidamente.
- **Resultados Empíricos:** Conforme visto na Seção 12.6, o speedup combinado no pré-processamento e avaliação da FO usando GPU (CuPy) atingiu $\approx 3{,}04\times$ em relação a uma implementação sequencial em CPU otimizada com NumPy. Tentativas de paralelização em CPU para as mesmas tarefas raramente ultrapassaram speedups de $2\times$ a $4\times$ (dependendo do número de cores e da natureza da tarefa), e frequentemente com maior complexidade de código para gerenciar processos e memória compartilhada. CuPy simplifica a escrita de código CUDA-like em Python.

13.6.3 Estratégias para Calibração de Penalidades Suaves (Soft Constraints)

A eficácia das restrições suaves depende crucialmente da calibração adequada dos coeficientes de penalidade M_1 (para violações de capacidade de carrinho, UB) e M_2 (para violações do limite de corredores, A). Nossa abordagem para calibrá-los envolveu:

- **Análise de Ordem de Grandeza:** As penalidades devem ser suficientemente grandes para desencorajar violações, mas não tão grandes a ponto de causar instabilidade numérica no solver ou de tornar o modelo intratável.
- M_1 (Capacidade): Ajustado em função do valor máximo possível da soma dos scores dos pedidos ($\sum_i s_i$). A ideia é que violar a capacidade UB por uma unidade seja sempre menos atraente (mais custoso) do que não incluir um pedido de score médio ou alto. Uma regra de ouro é que a penalidade deve ser maior que o benefício perdido ao não violar.
- M_2 (Corredores): Geralmente definido em uma ordem de grandeza inferior a M_1, para que o modelo priorize não violar a capacidade do carrinho em detrimento de violar o limite de corredores, se uma escolha tiver que ser feita. No entanto, ainda deve ser significativo o suficiente para penalizar o uso excessivo de corredores.
- **Testes de Sensibilidade:** Realizamos execuções experimentais variando M_1 e M_2 em faixas consideradas razoáveis (e.g., $[10^2, 10^5]$, dependendo da escala dos scores e custos implícitos). Observamos o impacto no valor da função objetivo, no número e magnitude das violações e no tempo de solução. Isso ajudou a identificar valores que proporcionam um bom equilíbrio entre flexibilidade, qualidade da solução e desempenho computacional. Para os resultados apresentados, utilizamos penalidades da ordem de 1000, que se mostraram robustas.

13.6.4 Garantia de Robustez e Consistência Numérica dos Resultados

Para assegurar a confiabilidade dos resultados e evitar inconsistências numéricas, adotamos as seguintes medidas:

- **Tolerâncias do Solver:** No CPLEX, configuramos tolerâncias de viabilidade e otimalidade relativamente estritas:

  - `FeasibilityTol = 1e-6` (tolerância para violação de restrições)
  - `OptimalityTol = 1e-6` (tolerância para o gap relativo de otimalidade, quando buscamos o ótimo exato)
  - `epgap = 0.01` (gap de 1% como critério de parada principal nos experimentos reportados)

  Isso garante que as soluções encontradas estejam dentro de limites de precisão matemática aceitáveis e que o status de "ótimo"seja confiável.

- **Convergência em Dinkelbach (Quando Usado):** Para os testes com Dinkelbach, o critério de convergência foi definido quando a diferença entre o valor da função objetivo fracionária e o parâmetro λ se tornava suficientemente pequena, por exemplo:

$$\Delta = \left| \frac{\sum_i s_i x_i}{1 + \sum_a y_a} - \lambda \right| \leq 10^{-4} \text{ ou } 10^{-5}.$$

- **Validação Cruzada e Testes de Sanidade:** Além das métricas quantitativas, realizamos verificações de sanidade nas soluções (e.g., garantindo que as restrições de capacidade e corredores não fossem violadas nas configurações com restrições rígidas, e que as penalidades fossem corretamente aplicadas nas configurações flexíveis). Os índices de corredores e a lógica de conflitos também foram validados com instâncias pequenas e casos de teste específicos.

Essas práticas contribuem para a robustez geral da solução e para a confiança nos resultados reportados.

14 Contribuições e Impacto Consolidados

Este trabalho transcende uma mera aplicação técnica, apresentando contribuições significativas e um impacto potencial considerável tanto para a academia quanto para a indústria. Respondemos diretamente às questões cruciais que especialistas e práticos levantam:

- **"Qual é o verdadeiro 'pulo do gato' desta abordagem?"**
- **"Quais são as provas concretas da eficácia do método proposto?"**
- **"Por que esta solução representa uma diferença substancial em relação ao que já existe?"**

14.1 O Que Este Trabalho Comprovadamente Demonstrou

1. **Viabilidade e Superioridade de Métodos Exatos para WOP em Escala Real:** Comprovamos que métodos exatos, especificamente a Programação Linear Inteira, podem ser notavelmente viáveis e eficientes para o Problema de Wave Order Picking (WOP) em cenários de grande escala (instâncias com até $n \approx 45000$ pedidos foram abordadas, e instâncias com $n \approx 300$ pedidos foram resolvidas otimamente em média em ≈ 32 segundos). A chave para este avanço reside na **aceleração massiva do pré-processamento de dados utilizando GPUs (CUDA/CuPy)** e na adoção de **modelos matemáticos flexíveis** (com restrições suaves). Isso desafia a noção anterior de que PLI seria intrinsecamente lento demais para aplicações práticas de WOP em tempo real ou quasi-real.
2. **PLIwC² Supera Heurísticas Consagradas em Qualidade e Compete em Tempo:** Nossa abordagem PLIwC² não apenas alcança soluções de qualidade superior (e.g., **91,2% do BOV oficial contra 75-88% de heurísticas como GRASP, SA e ILS**), mas o faz em tempos de execução altamente competitivos (e.g., ≈ 32 segundos para PLIwC² vs. 18-31 segundos para as heurísticas mencionadas). Em muitos casos, a pequena diferença de tempo é amplamente compensada pelo ganho substancial na qualidade da solução, que se traduz em maior eficiência operacional e econômica.

3. **Uma Pipeline Prática e Reproduzível (PuLP + CPLEX + CuPy):** Demonstramos a eficácia de uma pipeline que integra de forma inteligente a facilidade de modelagem do PuLP, o poder de resolução do CPLEX e a capacidade de computação paralela do CuPy. Esta arquitetura é **documentada e concebida para ser reproduzível**, servindo como um guia valioso para pesquisadores e desenvolvedores na indústria que buscam aplicar métodos exatos acelerados a outros problemas complexos.

14.2 Valor Agregado para a Academia e para a Indústria

- **Contribuição Acadêmica:** Este trabalho oferece um estudo de caso completo e detalhado da **sinergia entre Pesquisa Operacional exata e Computação de Alto Desempenho (HPC)** aplicado a um problema logístico relevante. Ele preenche uma lacuna ao demonstrar como técnicas de GPU podem ser integradas de forma eficaz em um fluxo de trabalho de otimização baseado em PLI, fornecendo insights e uma base para futuras pesquisas em otimizações híbridas CPU/GPU e algoritmos paralelos para otimização combinatória. O material suplementar (códigos, logs) serve como um tutorial prático.
- **Impacto Industrial:** Para a indústria de logística e e-commerce (como exemplificado pelo desafio do Mercado Livre ou operações da Amazon), a PLIwC^2 fornece uma **ferramenta de otimização de Wave Order Picking potencialmente transformadora**. A capacidade de obter soluções exatas ou muito próximas do ótimo, com gaps conhecidos e em tempos operacionalmente viáveis, permite um planejamento de coleta mais eficiente, redução de custos, melhor utilização de recursos (mão de obra e equipamentos) e, em última instância, maior satisfação do cliente. A flexibilidade do modelo também permite adaptar-se a diferentes políticas operacionais.
- **Impacto Social e Educacional:** O projeto serve como um excelente exemplo para a formação de estudantes de mestrado e graduação em áreas como Pesquisa Operacional, Ciência de Dados e Engenharia de Software, ilustrando a aplicação de conceitos teóricos em problemas do mundo real com tecnologia de ponta. Ele também inspira pesquisas futuras em áreas como decomposição de problemas, aprendizado de máquina para *warm-start* de solvers e otimização distribuída.

14.3 Reiteração do "Pulo do Gato": Aceleração CUDA no Pré-processamento

O diferencial fundamental da PLIwC^2 é a **vetorização completa e a execução em GPU de operações de pré-processamento que são computacionalmente intensivas e frequentemente quadráticas em complexidade** (como a construção da matriz de conflito pedido-pedido e a identificação de dominância), que tradicionalmente seriam gargalos em CPU. A redução do tempo de pré-solver de, por exemplo, ≈ 12 segundos (CPU) para $\approx 2{,}4$ segundos (GPU) resulta em um speedup global de aproximadamente $3\times$ nas fases que antecedem o solver. Essa economia de dezenas de segundos é crítica: ela é diretamente transferida para o CPLEX, permitindo-lhe explorar árvores de busca maiores, aplicar mais heurísticas e técnicas de corte, e, crucialmente, **aumentar a probabilidade de encontrar e provar a otimalidade dentro de janelas de tempo operacionais estritas**. Em muitas instâncias, a diferença entre uma solução otimizada com GPU (≈ 32 segundos) e uma solução apenas com CPU (≈ 70 segundos para o mesmo resultado de qualidade) pode ser o fator decisivo entre resolver o problema a tempo ou não.

14.4 Comparação Explícita com Soluções Mais Simples ou Alternativas

Para enfatizar o avanço, contrastamos diretamente:

- **PLI CPU-only (Nossa baseline otimizada sem GPU no pré-processamento):** Leva aproximadamente 70 a 120 segundos para instâncias de 200 pedidos. **PLIwC^2 com GPU:** ≈ 32 **segundos.**

- **Heurísticas Híbridas (e.g., mapeando para CVRP e resolvendo heuristicamente):** Frequentemente param em gaps de $\approx 15\%$ ou mais, ou exigem tempos ≥ 60 segundos para 100 pedidos, sem garantia de qualidade. **PLIwC²: gap $\leq 1\%$ em tempos comparáveis ou melhores para qualidade muito superior.**
- **Relaxação Lagrangeana:** Embora útil para obter limites, muitas vezes não converge para soluções primais de alta qualidade com gap $< 1\%$ em tempos práticos para instâncias de médio porte, especialmente quando comparada à capacidade da PLIwC² de provar otimalidade.

Logo, **não estamos apenas reinventando a roda com mais tecnologia**, mas sim **ampliando fundamentalmente o alcance e a eficácia de métodos exatos**: a integração *exato + GPU resulta em Função Objetivo* $\uparrow$ *e Tempo de Resolução* $\downarrow$, uma combinação poderosa.

14.5 Limitações Reconhecidas e Cenários de Uso Ótimos

Reiteramos as limitações para uma perspectiva equilibrada:

- **Requisitos de Hardware:** Uma GPU com ≥ 8 GB de VRAM é recomendada para instâncias maiores. Em ambientes de cluster compartilhado, a contenção por GPU (*thrashing*) pode aumentar o overhead de comunicação.
- **Escalabilidade para Instâncias Massivas ($>$ 500 a 1000 pedidos):** Para volumes de pedidos que excedem significativamente este porte, o modelo PLI monolítico pode se tornar intratável. Nesses casos, o pré-processamento GPU ainda é valioso, mas técnicas de *batching* adaptativo, decomposição (como Benders ou Dantzig-Wolfe), ou abordagens de coluna podem ser necessárias, elevando o tempo de pré-processamento (e.g., para $\approx 1{,}2$ s apenas para o pré-processamento de um batch, mas com a complexidade adicional da gestão dos batches).
- **Densidade do Modelo PLI:** Modelos muito densos (muitas interações entre variáveis) podem gerar matrizes de restrição que desafiam até mesmo solvers como o CPLEX, podendo exigir técnicas avançadas de warm-start ou cortes específicos do problema.

O cenário de uso ideal para a PLIwC² é onde a **alta qualidade da solução é crítica e os tempos de decisão são da ordem de minutos (e não milissegundos)**, para problemas de WOP de médio a grande porte (de dezenas a algumas centenas ou poucos milhares de pedidos, dependendo da estrutura).

14.6 Recomendações para Pesquisadores e Praticantes

- **Pesquisadores em WOP e Otimização Combinatória:** Encorajamos o download do repositório associado (se disponibilizado publicamente), a reprodução dos experimentos e, fundamentalmente, a adaptação das técnicas de pré-processamento vetorizado em GPU para outros problemas afins (e.g., scheduling com conflitos, packing com restrições de adjacência, VRP com janelas de tempo e interdependências).
- **Empresas com Sistemas de Gerenciamento de Transporte (TMS) ou Armazéns (WMS):** Sugerimos testar a integração do núcleo da PLIwC² (encapsulado como um microserviço, por exemplo, via `main.py` ou similar) em um ambiente de produção ou simulação com GPU dedicada. Avaliar o ganho de desempenho e qualidade *em campo* pode revelar economias e eficiências substanciais.
- **Professores de Pesquisa Operacional e Ciência de Dados:** Utilizar a Seção 11 e os detalhes do "Pulo do Gato"(Seção 13.5) como material didático para ilustrar a poderosa combinação de OR e HPC, inspirando uma nova geração de otimizadores.

15 Conclusão

Este trabalho apresentou a PLIwC², uma metodologia inovadora e de alto desempenho para o Problema de Wave Order Picking (WOP), que se destaca pela união estratégica de **modelagem exata baseada em Programação Linear Inteira Fracionária**, o uso inteligente de **restrições flexíveis (soft constraints)** para robustez e adaptabilidade, e, de forma crucial, um **pré-processamento de dados massivamente paralelizado e vetorizado em GPU (NVIDIA CUDA/CuPy)**. As conclusões fundamentais extraídas dos extensivos experimentos computacionais e análises são:

- **Exatidão Prático-Viável para WOP em Larga Escala:** Demonstramos conclusivamente que a PLIwC² é capaz de resolver instâncias complexas do WOP, incluindo aquelas com até aproximadamente 300 pedidos (e abordando até $N \approx 45000$), para otimalidade ou com gaps mínimos ($\leq 1\%$). Notavelmente, para a configuração otimizada (INVERSA+Flexível+GPU), instâncias de médio porte foram resolvidas em uma média de ≈ 32 **segundos**, alcançando uma qualidade de solução de $\approx 91,2\%$ **do Best Known Value (BOV)** oficial, um marco que redefine a aplicabilidade de métodos exatos neste domínio.
- **Eficácia Comprovada das Restrições Suaves:** A incorporação de *soft constraints* provou ser uma estratégia vital, não apenas para garantir a viabilidade em cenários altamente restritos, mas também para **acelerar a convergência do solver**. Esta abordagem elevou a taxa de instâncias resolvidas otimamente (e.g., de 18/20 para 20/20 no conjunto A dentro de 300s) e reduziu o tempo médio de solução em até 44%.
- **O "Pulo do Gato" – Pré-processamento Acelerado por GPU como Diferencial Competitivo:** A vetorização completa de operações computacionalmente intensivas no pré-processamento (como construção de matrizes de conflito e identificação de dominância), transferindo-as da CPU para a GPU, resultou em um **speedup global de aproximadamente** 3× nas fases que antecedem o solver. Esta aceleração é o principal fator que libera tempo valioso para o CPLEX, permitindo-lhe encontrar soluções de maior qualidade ou provar a otimalidade mais rapidamente.
- **Uma Pipeline Robusta e Reproduzível para Pesquisa e Indústria:** A arquitetura de software desenvolvida, integrando Python, PuLP, CPLEX e CuPy (detalhada na Seção 11), é modular e foi projetada para ser reproduzível, facilitando a validação por pares e a adaptação para outros problemas de otimização ou para integração em sistemas de produção industrial.
- **Impacto Acadêmico e Prático Significativo:** O trabalho não apenas avança o estado da arte na solução exata do WOP, mas também serve como um estudo de caso prático e um guia para a aplicação de técnicas de HPC (especificamente, computação em GPU) em problemas de Pesquisa Operacional. A PLIwC² se posiciona como uma **alternativa superior às heurísticas tradicionais** em termos de qualidade da solução, mantendo tempos de execução competitivos, o que tem implicações diretas para a eficiência e rentabilidade de operações logísticas.

Em suma, a PLIwC² demonstra que a fronteira do que é considerado "praticamente resolvível"por métodos exatos pode ser significativamente expandida através da aplicação inteligente de hardware moderno e técnicas de modelagem avançadas, oferecendo um caminho promissor para enfrentar desafios de otimização cada vez mais complexos no mundo real.

16 Trabalhos Futuros

Com base nos resultados promissores e nas lições aprendidas com o desenvolvimento da PLIwC², vislumbramos diversas direções férteis para investigações futuras, que podem expandir ainda mais a eficácia e o escopo de aplicação da abordagem:

1. **Hibridização Adaptativa com Heurísticas e Meta-heurísticas:** Investigar a criação de um framework híbrido inteligente que possa, dinamicamente, com base nas características da instância (tamanho,

densidade, etc.), decidir entre aplicar a PLIwC² para uma solução exata ou uma meta-heurística rápida (como ILS ou VNS) para obter uma solução inicial de alta qualidade (warm start) ou para lidar com instâncias excessivamente grandes onde o método exato puro ainda é inviável dentro de janelas de tempo muito curtas. A infraestrutura CUDA desenvolvida para o pré-processamento poderia ser alavancada também para acelerar componentes de meta-heurísticas.

2. **Exploração de Técnicas de Decomposição Avançadas:** Para instâncias de WOP verdadeiramente massivas (e.g., dezenas de milhares de pedidos com interações complexas), a abordagem monolítica do PLI pode atingir seus limites. A pesquisa sobre a aplicação de técnicas de decomposição, como Decomposição de Benders, Decomposição de Dantzig-Wolfe (Geração de Colunas), ou métodos baseados em Relaxação Lagrangeana com algoritmos de otimização de subproblemas mais sofisticados (e potencialmente acelerados por GPU), pode ser um caminho para melhorar a escalabilidade.

3. **Aprendizado de Máquina para Otimização (ML4CO):**

   - **Warm Start Inteligente:** Utilizar técnicas de Aprendizado de Máquina para prever valores iniciais promissores para as variáveis de decisão ou para o parâmetro λ no algoritmo de Dinkelbach, o que poderia acelerar significativamente a convergência.
   - **Seleção de Parâmetros Adaptativa:** Empregar ML para aprender a melhor configuração de parâmetros do solver (e.g., estratégias de branch, seleção de cortes) ou das penalidades M_1, M_2 com base nas características da instância.
   - **Poda de Árvore de Busca:** Explorar modelos de ML para guiar o processo de Branch-and-Bound, aprendendo políticas de poda de nós mais eficazes.

4. **Otimização Multi-Objetivo e Considerações Logísticas Adicionais:** Estender o modelo atual para um contexto multi-objetivo, considerando explicitamente outros fatores relevantes na operação logística, como o balanceamento de carga de trabalho entre coletores, a minimização do tempo total de percurso (TSP dentro da wave), a priorização de pedidos urgentes de forma mais granular, ou a minimização do congestionamento nos corredores.

5. **Aprimoramento Contínuo da Aceleração GPU:** Explorar primitivas CUDA mais avançadas ou bibliotecas especializadas para otimização combinatória em GPU que possam surgir. Investigar o uso de memória unificada para reduzir overheads de cópia CPU-GPU ou o processamento em GPU de partes do próprio algoritmo de Branch-and-Bound, se viável com as ferramentas atuais.

6. **Integração com Simulação e Análise de Robustez:** Combinar a PLIwC² com modelos de simulação de eventos discretos do armazém para validar o impacto das ondas geradas em um ambiente dinâmico e estocástico. Analisar a robustez das soluções frente a incertezas (e.g., variações no tempo de coleta, indisponibilidade de itens).

Acreditamos que a contínua exploração da sinergia entre otimização matemática rigorosa, algoritmos eficientes e o poder da computação paralela, especialmente em GPUs, continuará a expandir as fronteiras da otimização exata e aproximada para problemas complexos do mundo real, tornando-as ferramentas cada vez mais indispensáveis para a tomada de decisão inteligente.

17 Referências Bibliográficas

Mariano Almeida and Daniel Rebelatto. Princípios básicos para uma proposta de ensino sobre análise por envoltória de dados. Anais do XXXIV Congresso Brasileiro de Ensino de Engenharia, 2019. URL https://www.abenge.org.br/cobenge/legado/arquivos/13/artigos/14_285_716.pdf.

Ehsan Ardjmand, Heman Shakeri, Manjeet Singh, and Omid Sanei Bajgiran. Minimizing order picking makespan with multiple pickers in a wave picking warehouse. *International Journal of Production Economics*, 206:169–183, 2018. doi:10.1016/j.ijpe.2018.10.001.

Farid Azadivar and Hui Wang. A heuristic approach for the online order batching problem with multiple pickers. *Computers & Industrial Engineering*, 157:107322, 2021. doi:10.1016/j.cie.2021.107322.

Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi. Nemirovski. *Robust Optimization*. Princeton University Press, 1st edition, 2009. ISBN 978-0691135296.

Dimitris Bertsimas and Melvyn Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004. doi:10.1287/opre.1030.0065.

Dimitris Bertsimas and Melvyn Sim. *Robust Optimization*. SIAM, Philadelphia, 1st edition, 2011. ISBN 978-0898718823.

Rodrigo Boueri, Eduardo Rocha, and Fernando Rodopoulos. Análise envoltória de dados (dea) nas produções acadêmicas. In *XXVI Congresso Brasileiro de Custos*, pages 1–10, 2019. URL https://racef.fundace.org.br/index.php/racef/article/download/332/80.

Emrah Boz and Necati Aras. The order batching problem: A state-of-the-art review. *Sigma Journal of Engineering and Natural Sciences*, 40(2):402–420, 2022. doi:10.14744/sigma.2022.00018. URL https://dergipark.org.tr/en/download/article-file/2470183.

Abraham Charnes and William W. Cooper. Programming with linear fractional functionals. *Naval Research Logistics Quarterly*, 9(3-4):181–186, 1962. doi:10.1002/nav.3800090303. URL https://onlinelibrary.wiley.com/doi/10.1002/nav.3800090303.

Abraham Charnes, William W. Cooper, and Edward Rhodes. Measuring the efficiency of decision making units. *European Journal of Operational Research*, 2(6):429–444, 1978.

Tim Coelli, D.S. Prasada Rao, and George E. Battese. *An Introduction to Efficiency and Productivity Analysis*. Kluwer Academic Publishers, 1998.

Columbia University. 4.10 – the big m method, n.d. URL http://www.columbia.edu/~cs2035/courses/ieor3608.F05/david-bigM.pdf. Acesso em: 28 maio 2025.

Robert de Souza, Crispiniano de Jesus Gomes Furtado, and João Carlos Lopes Horta. Resolução gráfica de um problema de programação linear utilizando a folha gráfica 3d do geogebra. *Revista do Instituto GeoGebra de São Paulo*, 7(2):45–64, 2018. URL https://dialnet.unirioja.es/descarga/articulo/6767155.pdf. Disponível em: https://dialnet.unirioja.es/descarga/articulo/6767155.pdf.

W. Dinkelbach. On nonlinear fractional programming. *Management Science*, 13(7):492–498, 1967. doi:10.1287/mnsc.13.7.492. URL https://pubsonline.informs.org/doi/10.1287/mnsc.13.7.492.

Luís Henrique Rodrigues Duarte. Aplicação de métodos biobjetivo à otimização linear fracionária. Dissertação de mestrado em matemática, Universidade de Coimbra, Coimbra, Portugal, 2019. URL https://hdl.handle.net/10316/87827. Disponível em: https://hdl.handle.net/10316/87827 e https://estudogeral.uc.pt/bitstream/10316/87827/1/thesis.pdf.

Jonas Ferreira. *Modelos de análise envoltória de dados e suas aplicações*. PhD thesis, Universidade de São Paulo, 2015. URL https://www.teses.usp.br/teses/disponiveis/96/96132/tde-23102015-111731/publico/JonasFerreira_Corrigida.pdf.

Jia Gu, Mark Goetschalckx, and Leon F. McGinnis. Research on warehouse operation: A comprehensive review. *European Journal of Operational Research*, 203(3):539–549, 2010. doi:10.1016/j.ejor.2009.07.031.

Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com, 2023.

Sebastian Henn, Sören Koch, Karl F. Doerner, Christine Strauss, and Gerhard Wäscher. Metaheuristics for the order batching problem in manual order picking systems. *BuR - Business Research*, 3 (1):82–105, 2010. doi:10.1007/BF03342717. URL https://www.econstor.eu/bitstream/10419/103688/1/2508.pdf.

Sebastian Henn, Stefan Koch, and Gerhard Wäscher. Order batching in order picking warehouses: A survey of solution approaches. *International Journal of Production Research*, 50(3):779–802, 2012. doi:10.1080/00207543.2010.538744.

IBM. Ibm ilog cplex optimization studio. https://www.ibm.com/products/ilog-cplex-optimization-studio, 2017.

IBM Corporation. O que é modelagem de otimização?, 2023a. URL https://www.ibm.com/br-pt/think/topics/optimization-model.

IBM Corporation. *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*, 2023b. Versão 22.1, disponível em https://www.ibm.com/docs/en/icos/22.1.0.

Bernard Kolman. *Introdução à Programação Linear*. Editora XYZ, 1995.

René Koster, Theo Le-Duc, and Kees Jan Roodbergen. Design and control of warehouse order picking: A literature review. *European Journal of Operational Research*, 182(2):481–501, 2007. doi:10.1016/j.ejor.2006.07.009.

Hongxing Li. Linearization techniques for fractional programming with binary variables. *Operations Research Letters*, 15:231–238, 1994.

Yen-Chun Lin, Chih-Hung Wu, and Chia-Hsiang Chen. Wave planning for cart picking in a randomized storage warehouse. *Applied Sciences*, 10(22):8050, 2020. doi:10.3390/app10228050.

Wenrong Lu, Duncan McFarlane, Vaggelis Giannikas, and Quan Zhang. An algorithm for dynamic order-picking in warehouse operations. *European Journal of Operational Research*, 247(1):47–60, 2015. doi:10.1016/j.ejor.2015.05.019.

Sasan Mahmoudinazlou, Abhay Sobhanan, Hadi Charkhgard, Ali Eshragh, and George Dunn. Deep reinforcement learning for dynamic order picking in warehouse operations, 2024. Available at: https://arxiv.org/abs/2408.01656.

R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004. doi:10.1007/s00158-003-0368-6.

Francisco Daladier Marques Júnior. *Modelagem e Análise de Eficiência em Redes Virtuais Usando DEA e Modelos Fractais*. PhD thesis, Universidade Federal de Pernambuco, 2021. URL https://repositorio.ufpe.br/bitstream/123456789/35199/1/TESE%20Francisco%20Daladier%20Marques%20J%C3%BAnior.pdf.

Maxwell. Análise envoltória de dados. https://www.maxwell.vrac.puc-rio.br/16033/16033_3.PDF, 2014. Disponível em: https://www.maxwell.vrac.puc-rio.br/16033/16033_3.PDF, acesso em 30 maio 2025.

G. P. McCormick. Mccormick envelopes. https://optimization.cbe.cornell.edu/index.php?title=McCormick_envelopes. Acesso em: 02 jun 2025.

Garth P. McCormick. Computability of global solutions to factorable nonconvex programs. *Mathematical Programming*, 10:147–175, 1976.

Mercado Livre. Repositório do desafio no github. https://github.com/mercadolibre/challenge-sbpo-2025, 2025. Accessed: April 27, 2025.

Merve. Milp formulations for the order batching problem in low-level picker-to-part warehouse systems. Master of science thesis, Middle East Technical University, apr 2014. URL https://acikbilim.yok.gov.tr/bitstream/handle/20.500.12812/86973/yokAcikBilim_10035380.pdf?sequence=-1&isAllowed=y. Supervisor: Assoc. Prof. Dr. Temel "Oncan.

Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Springer, 1999. ISBN 978-0792374207.

MIT OpenCourseWare. *IP Reference guide for integer programming formulations*, 2013. URL https://ocw.mit.edu/courses/15-053-optimization-methods-in-management-science-spring-2013/86d472ffa3f1c341c586cb26ba1093c1_MIT15_053S13_iprefguide.pdf. Acesso em: 28 maio 2025.

Stuart Mitchell. Pulp documentation: Using cplex with pulp, 2024. URL https://coin-or.github.io/pulp/. Acessado em 4 de junho de 2025.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006. ISBN 978-0387303031.

Jean-François Pansart, Stéphane Dauzère-Pérès, and Michel Gourgand. A sparse milp formulation for the order batching problem in warehouses. *European Journal of Operational Research*, 271(3): 1076–1089, 2018. doi:10.1016/j.ejor.2018.06.049. URL https://doi.org/10.1016/j.ejor.2018.06.049.

João Pedro Pedroso. Método do big-m e suas aplicações em programação linear. Technical report, Universidade Federal de Santa Catarina, 2024. Material didático disponível em aula de Métodos de Apoio à Decisão.

Guilherme Fialho Costa Pocinho. *Análise e melhoria do processo de order-picking num sistema produtivo: caso de estudo*. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2013a. URL https://run.unl.pt/bitstream/10362/11038/1/Pocinho_2013.pdf.

Joana Pocinho. *Análise e Melhoria do Processo de order-picking num sistema de armazéns*. PhD thesis, Universidade Nova de Lisboa, 2013b. URL https://run.unl.pt/bitstream/10362/11038/1/Pocinho_2013.pdf.

PUC-Rio, Departamento de Engenharia Industrial. Métodos de otimização. https://www2.dbd.puc-rio.br/pergamum/tesesabertas/0310953_05_cap_03.pdf, 2011. Acesso em: 27 maio 2025.

Edison Rodrigues and Marina Silva. Otimização da eficiência operacional no processo de picking. *Revista do Encontro de Gestão e Tecnologia*, 12(2):45–56, 2022. URL https://revista.fateczl.edu.br/index.php/engetec_revista/article/view/193.

Kees Jan Roodbergen, Iris F. A. Vis, and Jan van den Berg. The integrated orderline batching, batch scheduling, and picker routing problem with multiple pickers. *Flexible Services and Manufacturing Journal*, 33:679–714, 2021. doi:10.1007/s10696-021-09425-8.

Siegfried Schaible. Fractional programming: Applications and algorithms. *European Journal of Operational Research*, 7(2):111–120, 1981. doi:10.1016/0377-2217(81)90272-0. URL https://www.sciencedirect.com/science/article/pii/0377221781902721.

Siegfried Schaible and Toshihide Ibaraki. Fractional programming. *European Journal of Operational Research*, 12(4):325–338, 1983. doi:10.1016/0377-2217(83)90153-9. URL https://www.sciencedirect.com/science/article/pii/0377221783901534.

Hanif D. Sherali and Warren P. Adams. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Springer, 1999.

Jiun-Yan Shiau and Jie-An Huang. Wave planning for cart picking in a randomized storage warehouse. *Applied Sciences*, 10(8050), 2020. doi:10.3390/app10228050.

Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1 edition, 1999. ISBN 978-0471359432.

Fengqi You and Ignacio E. Grossmann. Mixed-integer programming models and algorithms for integrated capacitated production planning and scheduling in continuous manufacturing industries. *Computers & Chemical Engineering*, 33(12):1711–1727, 2009. doi:10.1016/j.compchemeng.2009.02.005. URL https://www.sciencedirect.com/science/article/pii/S0098135409000551.

T. Öncan. Mathematical models for order batching in warehouses. *International Journal of Production Research*, 53(11):3388–3402, 2015. doi:10.1080/00207543.2014.993958. URL https://doi.org/10.1080/00207543.2014.993958.