Dissertação de Mestrado

Uma meta-heurística para o Problema da Mochila com Penalidades

Matheus Machado Vieira
mmv@ic.ufal.br

Orientadores:

Prof. Dr. Rian Gabriel Santos Pinheiro
Prof. Dr. Bruno Costa e Silva Nogueira

Maceió, 16 de dezembro de 2023

Matheus Machado Vieira

Uma meta-heurística para o Problema da Mochila com Penalidades

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Rian Gabriel Santos Pinheiro

Prof. Dr. Bruno Costa e Silva Nogueira

Maceió, 16 de dezembro de 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

V658m Vieira, Matheus Machado.
Uma meta-heurística para o problema da mochila com penalidades / Matheus Machado Vieira. – 2023.
45 f. : il.

Orientador: Rian Gabriel Santos Pinheiro.
Co-orientador: Bruno Costa e Silva Nogueira.
Dissertação (mestrado em informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2023.

Bibliografia: f. 40-45.

1. Otimização combinatória. 2. Problema da mochila. 3. Meta-heurísticas. 4. Busca local iterada. 5. Descida em vizinhança variável. I. Título.

CDU: 004.023

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
Av. Lourival Melo Mota, S/N, Tabuleiro do Martins, Maceió - AL, 57.072-970
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO (PROPEP)
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Folha de Aprovação

MATHEUS MACHADO VIEIRA

UMA META-HEURÍSTICA PARA O PROBLEMA DA MOCHILA COM PENALIDADES

METAHEURISTICS FOR THE KNAPSACK PROBLEM WITH FORFEITS

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 31 de janeiro de 2024.

Banca Examinadora:

Documento assinado digitalmente
gov.br
RIAN GABRIEL SANTOS PINHEIRO
Data: 06/02/2024 11:17:10-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. RIAN GABRIEL SANTOS PINHEIRO
UFAL – Instituto de Computação
Orientador

Documento assinado digitalmente
gov.br
BRUNO COSTA E SILVA NOGUEIRA
Data: 06/02/2024 14:15:53-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. BRUNO COSTA E SILVA NOGUEIRA
UFAL – Instituto de Computação
Coorientador

Documento assinado digitalmente
gov.br
ERICK DE ANDRADE BARBOZA
Data: 06/02/2024 11:48:51-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. ERICK DE ANDRADE BARBOZA
UFAL – Instituto de Computação
Examinador Interno

Documento assinado digitalmente
gov.br
DIMAS CASSIMIRO DO NASCIMENTO FILHO
Data: 07/02/2024 14:26:25-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. DIMAS CASSIMIRO DO NASCIMENTO FILHO
Universidade Federal do Agreste de Pernambuco-UFAPE
Examinador Externo

Resumo

O problema da mochila está entre um dos problemas combinatórios mais conhecidos. Seu potencial de aplicação e as inúmeras variações que existem fazem dele um bom modelo para diversos problemas práticos da vida real. Mais especificamente, este trabalho aborda uma variante do problema, o Problema da Mochila com Penalidades (PMP, ou, em inglês, KPF). Nesta variante, é fornecido um conjunto de itens e um grafo de conflitos, e o objetivo é identificar uma coleção de itens que respeite a capacidade da mochila enquanto maximiza o valor total dos itens menos as penalidades pelos itens conflitantes. O PMP tem tido algum engajamento tanto por sua proximidade com outros problemas famosos, como o do conjunto independente de peso máximo, e suas aplicações. Alguns exemplos compreendem desde a organização da força de trabalho em chão de fábrica até problemas de decisão em investimentos. Este trabalho apresenta um novo método para o problema utilizando-se de ferramentas já bem estabelecidas, baseado na hibridização de *Iterated Local Search* (ILS), *Variable Neighborhood Descent* (VND), e elementos de *Tabu Search*. Nosso método leva em consideração quatro estruturas de vizinhança, introduzidas com estruturas de dados eficientes para explorá-las. Resultados experimentais demonstram que a abordagem proposta supera os algoritmos de ponta na literatura. Em particular, o método proposto fornece soluções superiores em tempos de computação significativamente mais curtos em todas as instâncias de referência. Também foi incluída uma análise de como as estruturas de dados propostas influenciaram tanto a qualidade das soluções quanto o tempo de execução do método.

Palavras-chave: Otimização Combinatória, Problema da Mochila, Meta-heurísticas, Busca local iterada, Descida em vizinhança variável.

Abstract

The knapsack problem is among one of the most well-known combinatorial problems. Its potential for application and the numerous variations that exist make it a good model for various practical real-life problems. More specifically, this work addresses a variant of the problem, the Knapsack Problem with Forfeits (KPF). In this variant, a set of items and a conflict graph are provided, and the goal is to identify a collection of items that respects the knapsack's capacity while maximizing the total value of the items minus the penalties for conflicting items. The KPF has garnered some attention both due to its proximity to other famous problems, such as the maximum weight independent set, and its applications. Some examples range from workforce organization on the factory floor to decision problems in investments. This work presents a new method for the problem using well-established tools, based on the hybridization of *Iterated Local Search* (ILS), *Variable Neighborhood Descent* (VND), and elements of *Tabu Search*. Our method takes into account four neighborhood structures, introduced with efficient data structures to explore them. Experimental results demonstrate that the proposed approach outperforms state-of-the-art algorithms in the literature. In particular, the proposed method provides superior solutions in significantly shorter computation times on all reference instances. An analysis of how the proposed data structures influenced both the quality of the solutions and the method's execution time is also included.

Keywords: Combinatorial optimization, Knapsack problem, Metaheuristics, Iterated Local Search, Variable Neighborhood Descent.

Sumário

1 Introdução

O problema da mochila por si só é, na verdade, uma vasta classe de problemas de decisão de subconjuntos de itens. O problema consiste em selecionar um conjunto de itens com valores e pesos atribuídos com o objetivo de maximizar a soma dos valores restringindo a soma dos pesos a um valor pré-estabelecido. Outras restrições podem ser adicionadas, como a proibição de certos pares (HIFI; OTMANI, 2012), ou apenas os penalizando (CERULLI et al., 2020).

O problema em sua forma mais básica (Mochila 0-1) começou a ganhar atenção a partir de 1950 (SALKIN; KLUYVER, 1975), sendo encontrado em problemas de manejo de capital, carregamento de produtos, seleção de projeto entre outros. Nessa mesma época se desenvolvia melhor os conceitos de programação dinâmica (BELLMAN, 1954) e otimização inteira (BRADLEY, 1971). Já as extensões do problema que abraçam as restrições de proibição ou penalização são um pouco mais recentes (YAMADA; KATAOKAT; WATANABET, 2002; CERULLI et al., 2020).

A versão com penalidades do problema pode ser facilmente representada por um grafo. Usando a notação usual, o problema pode ser apresentado como um grafo de conflitos $G = (V, E)$ onde, neste caso, as arestas possuem pesos que representam penalidade para certos pares de itens presentes na solução, e os vértices possuem tanto um peso quanto um valor, representando o item de fato. O problema estão se torna buscar um subgrafo cuja soma dos valores dos vértices subtraída da soma dos valores das arestas seja a maior possível, respeitando que a soma dos pesos dos vértices seja menor que um valor pré-estabelecido (capacidade da mochila). Se considerarmos que os conflitos são de fato impraticáveis (seja por uma proibição de fato ou por ter um custo que o torne inviável), temos outra variação do problema (que acaba sendo um caso especial), desta vez considerando apenas que arestas não podem estar presentes na solução, sendo este mais estudado na literatura sendo chamado de Problema da Mochila Disjunta. Enquanto isso, a versão do problema com penalidades é a mais abrangente entre as citadas anteriormente, uma vez que todos os outros problemas podem ser vistos como casos especiais. Dessa forma, é possível encontrar situações reais que podem ser modeladas por esse problema ou os problemas similares a esse.

Dentre aplicações, é possível imaginar situações onde o transporte de determinados itens que, idealmente, não deveriam estar juntos, como em transporte de químicos. Ou ainda a seleção de medicamentos para um tratamento de larga escala com orçamento limitado, onde a métrica seria a efetividade do conjunto de medicamentos escolhidos a menos das possíveis interferências que possam causar entre si. Também pode ser considerado a escolha de ativos produtivos, visando maximizar a produtividade e restrito a um orçamento, com as penalidades representando a redução da produtividade, uma vez que ambos os itens do par geram o mesmo produto, assim não sendo mais tão produtivo quanto poderia. Mais alguns casos de uso podem ser citados, como o problema de entrega de correspondência em zonas rurais (COLOMBI et al., 2017). Neste problema, entregas a partir de um depósito devem ser feitas e existem caminhos para isto, caminhos que podem ser rentáveis, outros que podem gerar um ganho apenas se um determinado custo for pago, ou ainda outros que são completamente inviáveis. Ou o problema com operação de maquinário (CERULLI et al., 2020) em que se considera linhas paralelas de produção e a alocação de operários, onde possivelmente existem conjuntos de operários que quando alocado juntos causam um gasto extra, mas que acabam gerando um ganho maior do que quando não considerados. Outros problemas similares incluem outros tipos de seleção, como a maximização de tempo operacional de redes de sensores (CERULLI et al., 2022; CARRABS et al., 2017).

É possível encontrar o Problema da Mochila com Penalidades (PMP), por exemplo, em situações onde as restrições do problema com proibições pode ser relaxado para penalidades. Podemos considerar o seguinte exemplo: Dado um conjunto de tarefas que pode ser executada, se deseja escolher um subconjunto dessas tarefas que, novamente, não ultrapassem um orçamento e que levem a um ganho máximo. As restrições de pares entram quando consideramos que duas tarefas podem requerer o mesmo executor. Ao observar apenas isso, temos uma proibição nas restrições, mas, se existir a possibilidade de duplicar o executor por um certo custo, isto pode ser considerado uma penalidade e pode selecionar ambas tarefas. Mas, ainda sobre a versão do problema com proibições, ele é mais facilmente relacionável com outros problemas de otimização combinatórios, como *bin packing* (MURITIBA et al., 2010; SADYKOV; VANDERBECK, 2013), escalonamento (BODLAENDER; JANSEN, 1993; BAKER; JR, 1996), arvores geradoras (DARMANN et al., 2011) e fluxo máximo (PFERSCHY; SCHAUER, 2013).

O PMP possui um grande domínio de aplicação, visto que é um problema mais abrangente que possui casos específicos amplamente estudados na literatura, como os anteriormente citados. Isto torna os métodos aplicados ao problema também generalistas, embora ainda se tenha pouco material disponível sobre o problema na literatura por ser recente e ter recebido pouca atenção até o momento.

Também podem ser citados outros problemas semelhantes ao PMP, mas não diretamente aplicáveis, como, por exemplo, o Problema da Mochila com Penalidade (CESELLI; RIGHINI, 2006a; Della Croce; PFERSCHY; SCATAMACCHIA, 2019), no singular, onde, cada item possui um valor e uma penalidade, e dentre os itens escolhidos, a penalidade máxima entre eles é paga. Tem-se também o problema da Mochila com Custo Fixo (YAMADA; TAKEOKA, 2009; AKINC, 2006), onde os itens são arranjados em partições, e cada partição tem um custo a ser pago se pelo menos um item que faz parte dela for escolhido para a solução.

Já é sabido que o problema 0-1 é NP-difícil (PISINGER, 2005), e por isso suas generalizações também são. Isto pode facilmente ser visto quando consideramos uma instância do PMP cujo conjunto de penalidades é vazio, sendo assim uma instância do problema 0-1. Portanto, a depender do número de itens em uma instância, métodos exatos têm dificuldade em encontrar soluções em tempo aceitável. Mas ainda assim, métodos de programação dinâmica (AXIOTIS; TZAMOS, 2018) e *branch-and-bound* (CONIGLIO; FURINI; SEGUNDO, 2021), por exemplo, continuam a ser desenvolvidos e trazer resultados.

Recentemente, houve o desenvolvimento de métodos para a resolução do problema penalizado, seguindo por diferentes abordagens. A meta-heurística do Carrossel Guloso (CERRONE; CERULLI; GOLDEN, 2017) foi aplicada juntamente à formalização do problema da mochila com essas restrições (CERULLI et al., 2020). Também foram aplicados métodos baseados em algoritmos genéticos como o Genetic Algorithm-Carousel Greedy(GA-CG) (CAPOBIANCO et al., 2021) e MemeticFS (MA) (D'AMBROSIO et al., 2023). Este ultimo foi desenvolvido para uma generalização do PMP, mas também foi aplicado ao problema em questão, trazendo bons resultados e sendo considerado a melhor opção para o problema no momento.

Para o problema da mochila com proibições (ou disjuntivamente restrito) (YAMADA; KATAOKAT; WATANABET, 2002; SALEM et al., 2018), o progresso vem sendo feito, em parte, devido à grande proximidade entre este problema e problemas clássicos de clique (BOMZE et al., 1999) e conjuntos independentes (NOGUEIRA; PINHEIRO; SUBRAMANIAN, 2017). Este trabalho se propõe a estudar soluções para o PMP (CERULLI et al., 2020) e apresentar um método desenvolvido para a resolução do mesmo. O método proposto tem como foco o desenvolvimento de um método ILS-VND baseado em outro

já aplicado com sucesso em um problema similar (NOGUEIRA; PINHEIRO; TAVARES, 2020).

O método proposto neste trabalho toma como entrada uma instância do PMP e gera uma solução inicial a partir de uma heurística gulosa, qual é submetida a um processo iterativo, onde se é aplicada perturbações e buscas locais. Em cada iteração, os ótimos locais alcançados são avaliados, substituindo o ótimo local anterior quando possui um maior valor com auxílio de um mecanismo de intensificação e diversificação das soluções. A esse processo iterativo se dá o nome de *Iterated Local Search* (ILS).

A busca local consiste na execução de dois processos iterativos, ambos implementados como buscas tipo *Variable Neighborhood Descent* (VND). A primeira busca considera dois operadores de troca de itens, o `swap-21`, que troca dois itens da solução por um de fora, e `swap-11`, que troca um de dentro por um de fora. Já a segunda busca trabalha com operadores simples sobre a solução, inserção e remoção. Todos os operadores capazes de remover itens da solução o fazem obedecendo condições de "tempo de permanência" impostas por uma Lista Tabu.

A qualidade do algoritmo desenvolvido foi mensurada por meio de experimentos e comparações com os outros algoritmos já aplicados. Como resultado, apresentamos este novo método de resolução e novas contribuições para o melhor entendimento do problema.

1.1 Objetivos

Nesta seção, são apresentados os objetivos deste trabalho, onde se buscou implementar e analisar um novo método, como também compará-lo com métodos de resolução para o PMP já existente na literatura. Para isto:

- Propor e implementar um método ILS-VND eficiente para a resolução do Problema da Mochila com Penalidades.
- Observar os componentes (operadores, parâmetros, etc.) do método construído de modo a identificar os aspectos que o tornem eficiente.
- Desenvolver uma busca local de baixa complexidade que possa ser facilmente adaptada para outros problemas.
- Propor e implementar estruturas de dados eficientes à forma como o método é estruturado.
- Comparar o método proposto com métodos do estado-da-arte (GA-CG e MA), considerando instâncias conhecidas.

1.2 Estrutura do trabalho

Os próximos capítulos deste trabalho são organizados da seguinte forma: uma fundamentação teórica (Capítulo 2) trazendo alguns dos conhecimentos necessários para um bom entendimento deste trabalho, uma revisão da literatura (Capítulo 3) com alguns dos métodos de resolução do problema abordado e seus relacionados, a apresentação do método construído (Capítulo 4), resultados obtidos (Capítulo 5), considerando a evolução durante o desenvolvimento e análises dos componentes do método. Por fim, conclusões e considerações finais (Capítulo 6).

2 Fundamentação Teórica

Neste capítulo serão apresentados conhecimentos prévios necessários para a melhor compreensão da pesquisa. Serão descritos os termos que estabelecem os conceitos de otimização, NP-completude, problemas combinatórios e métodos de resolução.

As apresentações sobre otimização e NP-completude serão dadas em termos mais gerais, enquanto os demais tópicos serão dedicados à natureza matematicamente discreta de alguns problemas, os quais também serão apresentados.

2.1 Otimização

A idealização do conceito de otimização esteve presente na sociedade desde que problemas sobre como melhorar processos foram pensados, sejam problemas estes com ou sem restrições, com aplicações gerais ou unicamente específica. Essa melhoria buscada pode vir das mais diversas formas, a depender do problema, como redução do tempo de produção de itens manufaturados, ou, na melhor distribuição de recursos para alguma determinada atividade. Mais notoriamente, o uso de uma matemática mais avançada para melhoras de problemas pode ser rastreada facilmente até o Século XVIII, pelo já conhecido nome de Babbage para problemas de correspondência (HOPKINS, 2021), passando por Kantorovich (KANTOROVICH, 1960) e Dantzig (DANTZIG, 1963), no contexto das duas grandes guerras, mais precisamente sobre o funcionamento de maquinários e suprimentos.

De forma generalista, é possível considerar otimizável algo que possa ser medido. Portanto, pode ser definido um domínio $I \subseteq R^n$ das possibilidades do problema de onde se deseja encontrar o melhor arranjo possível. Como temos medidas atribuídas aos possíveis arranjos, pode ser definida uma função $f : I \to \mathbb{R}$ que concretize essa medida. Assim, um ótimo de f em I é um ponto x^* que satisfaça:

1. $f(x^*) \leq f(x), \forall x \in I$, ou;
2. $f(x^*) \geq f(x), \forall x \in I$.

Sendo o primeiro caso um ponto de valor mínimo, e o segundo, de valor máximo. O que se é satisfeito depende da situação, como quando em um problema de redução de custos o ótimo é o mínimo, ou quando em um problema de ganhos o ótimo é o máximo.

É imperativo considerar as restrições que desempenham um papel crucial na determinação da viabilidade de soluções ótimas. Restrições na otimização são, essencialmente, funções ou condições que definem os limites dentro dos quais as soluções devem residir para serem consideradas viáveis. Elas podem ser representadas por igualdades ou desigualdades que restringem o conjunto de soluções possíveis a um subconjunto do domínio original I, ou ainda por restrições de domínios de certos elementos da instância. Por exemplo, restrições de não-negatividade em variáveis de decisão, limitações de capacidade, ou exigências de quantidades exatas.

Ainda podemos considerar duas caracterizações básicas para problemas de otimização, problemas contínuos e discretos. Um problema continuo tem como atributo seu conjunto I ser também contínuo, o que, em termos gerais, nos permite escolher uma variação infinitesimal para qualquer arranjo. Enquanto em um problema discreto, temos, pelo menos, um conjunto enumerável de elementos para um $I \subseteq \mathbb{Z}^n$. No conjunto de problemas discretos, ainda podemos considerar o caso que I é um conjunto finito. Estes problemas são normalmente chamados de problemas de otimização combinatória, e sua representação mais usual se dá por meio de grafos.

2.2 Complexidade

Dado um problema de otimização e seu conjunto de arranjos possíveis, existe uma segunda questão inerente ao problema, a qual considera o tempo disponível para encontrar tal solução ótima. Ao pensar no tempo que um computador leva para resolver um problema, é comum considerar o tempo de cada operação feito no processo de solução sobre a entrada do problema. Para um problema simples, como o de encontrar uma permutação ótima para um conjunto, pode ser determinado o tempo máximo de busca com facilidade. Para isto, basta considerar o produto do tempo de verificação de uma permutação com o número de permutações possíveis. Esse conceito de tempo de execução máximo pode ser aplicado para qualquer método de solução (HARTMANIS; STEARNS, 1965). Para o problema de permutações, é perceptível que o tempo de execução pode ser limitado superiormente por uma função exponencial, visto que o número de permutações é dado também por uma função exponencial.

A função que limita superiormente o tempo de execução de um problema, em outras palavras, o tempo máximo de execução possível, ou pior caso, possui o poder de determinar a tratabilidade do problema em questão a partir de seu crescimento (consoante com o tamanho ou dimensionalidade do conjunto de entrada). Existem dois grandes conjuntos de funções consideradas para o tempo de execução, ou complexidade computacional (COOK, 2000).

O primeiro conjunto (comumente chamado de P) é constituído por problemas cujo tempo máximo para encontrar a solução pode ser descrita como uma função polinomial cuja entrada é o tamanho da instância. O segundo conjunto, NP, é formada pelos problemas cujas soluções podem ser verificadas em tempo polinomial.

Dessa forma, os problemas podem herdar a denominação de suas funções de tempo de execução, daí surgem as denominações de problemas pertencentes à classe P, ou problemas pertencentes à classe NP, dos problemas que podem ter uma solução gerada e verificada em tempo polinomial. É válido citar que todo problema P também é NP, mas não se sabe o contrário.

Estudos sobre a complexidade computacional mostram certas características pertinentes a esses conjuntos de problemas. Cook (COOK, 1983), Karp (KARP, 1972) e Hartmanis (HARTMANIS, 1982) são nomes clássicos no meio. Mais precisamente, Cook e Karp demonstraram a redutibilidade de alguns problemas NP a outros. Se todos os problemas em NP puderem ser transformados em tempo polinomial em um problema p, é dito que p é um problema NP-completo.

O impacto da demonstração dos problemas NP-completos vem do fato de que, se todo problema NP pode ser transformado (ou reduzido) em um problema NP-completo em tempo polinomial, e existe pelo menos um método de solução em tempo polinomial para qualquer problema NP-completo, implica em dizer que qualquer problema em NP possui um algoritmo em tempo polinomial, ou ainda, $P = NP$, o que é um dos problemas abertos na área até atualmente. Ainda existem problemas NP-difíceis, que para qualquer problema NP, existe uma transformação em tempo polinomial para um NP-difícil. Portanto, todo problema NP-completo também é NP e NP-difícil, enquanto um problema NP-difícil não necessariamente está em NP.

2.3 Grafos

Os grafos são uma representação convencional para problemas combinatórios (AVIS; HERTZ; MARCOTTE, 2005). Devido a sua estrutura, elementos discretos (vértices) de um determinado conjunto podendo ou não ter relação entre si (arestas), os grafos possuem um papel fundamental no meio da otimização combinatória no sentido de que

diversos métodos de resolução se apoiam na teoria de grafos para seu funcionamento. Uma definição formal para um grafo $G = (V, E)$ é uma tupla em que:

- V é um conjunto elementos discretos, vértices;
- $E \subseteq V \times V$, é o conjunto de relações, arestas.

Também se pode citar alguns problemas já conhecidos sobre grafos que tem alguma relação com o problema estudado. Dentre eles, podemos citar o problema da clique máxima (BOMZE et al., 1999), grafo de conflitos (PFERSCHY; SCHAUER, 2009), o problema do conjunto máximo independente e sua generalização (COLOMBI; MANSINI; SAVELSBERGH, 2017), com este último podendo ser aplicado em problemas de colheita florestal (HOCHBAUM; PATHRIA, 1997) até posicionamento de etiqueta cartográfica (MAURI; RIBEIRO; LORENA, 2010).

2.3.1 Conjunto Máximo independente

Uma melhor representação em um grafo para o problema da mochila quando se é inclusa a restrição de que proíbe certos pares de itens (Seção 2.4.2) é considerar essa restrição como uma aresta no grafo. Em outras palavras, com um vértice sendo um item, uma aresta entre dois vértices significa dizer que os dois itens representados pelos vértices não podem estar na mochila ao simultaneamente. Podemos então representar o problema da mochila disjunta em um grafo com essa configuração, como sendo o problema de buscar um subconjunto de vértices que não tenham arestas entre si, e, que o valor somado dos vértices desse subconjunto seja o maior possível. A partir daí temos o problema comumente chamado de conjunto de peso máximo independente (TARJAN; TROJANOWSKI, 1977):

$$\begin{aligned} \max \quad & \sum_{v \in C} w_v \\ \text{s. t.} \quad & C \subseteq V \\ & (i,j) \notin E \qquad\qquad \forall i, j \in C, i \neq j \end{aligned}$$

Para outra estrutura similar de grafos, cliques (em que se considera o inverso, que todos os vértices devem estar conectados entre si) (BOMZE et al., 1999), existe uma relação íntima a ponto de serem complementares. Dado um grafo $G = (V, E)$, o seu complementar $\overline{G}$ é outro grafo que contém os mesmo vértices, e que as arestas são aquelas que não existem em E. Em outras palavras, $\overline{G} = (V, \overline{E})$, onde:

$$\overline{E} := \{(i,j) | \forall i, j, i \neq j, (i,j) \notin E\}.$$

Por essa definição, é possível perceber que uma clique em um grafo G é um conjunto independente em $\overline{G}$. Portanto, uma vez que seja possível resolver o problema do conjunto de peso máximo independente por algum método, é possível utilizar-se desse mesmo método para resolver o problema da clique de peso máximo, bastando considerar o grafo complementar ao inicial.

A mesma ideia pode ser aplicada para o problema da mochila disjunta, que do ponto de vista teórico, é o problema do conjunto independente de peso máximo generalizado, permitindo assim que seja resolvido tanto por métodos para cliques quanto para conjuntos independentes.

Uma vez mostrada a relação entre os dois problemas, o da mochila e conjunto máximo independente, é possível perceber que o grafo tem como clara característica ser um grafo

de conflito, por motivos óbvios. A resolução de grafos de conflito (não necessariamente no contexto apresentado) permite diversas formas de resolução, desde o estudo de árvores (PFERSCHY; SCHAUER, 2009) até otimização inteira (ATAMTüRK; NEMHAUSER; SAVELSBERGH, 2000).

2.3.2 Conjunto Máximo independente generalizado

O problema do conjunto máximo independente pode ser ainda mais generalizado. O problema inicialmente considera que nenhuma aresta pode estar na solução. No entanto, podemos considerar dois conjuntos de arestas, um que não devem estar na solução (E_1) e outro dos que podem estar presentes E_2. Este segundo conjunto de arestas ainda pode ter um valor associado para cada par vértice que relaciona (c_{ij}), servindo como uma penalidade. De certa forma, pode ser considerado que existem dois tipos de conflitos, um que torna a solução inviável e outro que apenas penaliza a solução.

Essa generalização é conhecida como o problema do conjunto independente generalizado (em inglês, GISP) (COLOMBI; MANSINI; SAVELSBERGH, 2017) e até então pouco estudada. Para definirmos com precisão o problema, é necessário ter mais controle sobre a estrutura, complementando a estrutura já apresentada até agora:

$$
\begin{aligned}
\max \quad & \sum_{v \in V} x_v w_v - \sum_{(i,j) \in E_2} y_{ij} c_{ij} & \\
\text{s. t.} \quad & x_i + x_j \leq 1, & \forall (i,j) \in E_1 \\
& x_i + x_j - y_{ij} \leq 1, & \forall (i,j) \in E_2 \\
& x_v \in \{0,1\}, & \forall v \in V \\
& y_{ij} \in \{0,1\}, & \forall (i,j) \in E_2
\end{aligned}
$$

onde $x_v = 1$, quando $v \in V$ está na solução e $x_v = 0$ quando não, e, $y_{ij} = 1$ quando $(i,j) \in E_2$ estão na solução. Rapidamente podemos perceber que o GISP é o problema mais próximo aqui apresentado ao PMP, basta que a restrição de capacidade da mochila seja adicionada e que $E_1 = \varnothing$.

O GISP, por estar em níveis mais altos de generalizações, podendo representar problemas de cliques e conjuntos independentes de formas mais gerais, possui um imenso potencial de aplicações, como colheita florestal (HOCHBAUM; PATHRIA, 1997), manejo de incertezas geográficas em informações espaciais (WEI; MURRAY, 2012), localização de instalações (HOCHBAUM, 2004) e posicionamento de etiqueta cartográfica (MAURI; RIBEIRO; LORENA, 2010). Além disso, podemos observar que também faz parte dos problemas NP-difíceis.

2.4 Problema da mochila

O problema da mochila (PISINGER; TOTH, 1998) é um dos problemas combinatórios mais estudados, seja por sua simplicidade ou por ser um subproblema recorrente. De forma geral, consiste em selecionar um conjunto de itens para maximizar o valor contido de dentro da "mochila" com os itens escolhidos, com uma condição de que o peso máximo da "mochila" deve ser respeitado. Em outras palavras, é selecionar o subconjunto de itens mais valioso cujo peso é menor ou igual à capacidade da mochila. Existem também outros tipos de problemas da mochila, seja considerando múltiplas mochilas, com conjuntos de itens proibidos (PFERSCHY; SCHAUER, 2009) ou penalizados (CERULLI et al., 2020).

O problema pertence à classe NP-difícil, e por isso é um problema recorrentemente abordado em termos de otimização combinatória. Portanto, para grandes instâncias de

problemas, o uso de métodos exatos para o problema acaba se tornando inviável. Suas aplicações vão desde seleção de cargas, decisão em um contexto de investimentos, roteamento de veículos, e até criptografia (DIFFIE; HELLMAN, 2019).

2.4.1 Mochila 0-1

Considerando-se uma visão ampla de décadas atrás (SALKIN; KLUYVER, 1975), é perceptível o grande interesse na forma mais básica do problema, comumente conhecido como o problema da Mochila 0-1. Considerando um conjunto de itens disponíveis, denotado por X, onde p_i e w_i representar o valor e o peso, respectivamente, do item i, para todo $i \in X$, e seja b a capacidade máxima da mochila. Com essas definições, o problema da Mochila 0-1 pode ser formulado como o seguinte modelo de programação inteira:

$$\max \quad \sum_{i \in X} p_i x_i \tag{1}$$

$$\text{s. t.} \quad \sum_{i \in X} w_i x_i \leq b \tag{2}$$

$$x_i \in \{0, 1\} \qquad \forall i \in X \tag{3}$$

Onde as variáveis de decisão $x_i, \forall i \in X$, são binarias, indicando quando o item i é selecionado para estar na solução ($x_i = 1$) ou não ($x_i = 0$). Em outras palavras, o problema consiste em encontrar o subconjunto de elementos de maior valor entre todos os possíveis subconjuntos que não excedem a capacidade da "mochila".

Uma característica interessante para ser apresentada sobre o problema é que existe a possibilidade de transformação de problemas de otimização inteira para o problema da mochila (MATHEWS, 1896). Dada a simplicidade do problema, isso pode trazer melhor compreensão para a natureza do problema de otimização inteira original, podendo até trazer formas de resolução mais eficientes.

2.4.2 Mochila disjunta

Uma das possíveis formulações para o problema da mochila é o problema disjunto (PFERSCHY; SCHAUER, 2009). Esse novo problema é uma versão mais generalizada do problema clássico 0-1 onde existem certos pares de itens que não podem estar presentes na solução. Dessa forma, a definição do problema da Mochila 0-1 pode ser estendida, adicionando à todas as definições já feitas uma nova: um conjunto $E \subseteq X \times X$, representando os pares proibidos (i, j), onde $i, j \in X$, sendo estes os quais não podem estar presentes na solução simultaneamente. Como resultado se tem o seguinte modelo:

$$\max \quad \sum_{i \in X} p_i x_i \tag{4}$$

$$\text{s. t.} \quad \sum_{i \in X} w_i x_i \leq b \tag{5}$$

$$x_i + x_j \leq 1 \qquad \forall (i, j) \in E \tag{6}$$

$$x_i \in \{0, 1\} \qquad \forall i \in X \tag{7}$$

As novas restrições (6) é o que garante que ambos os itens não estarão na solução simultaneamente, uma vez que $1 + 1 \leq 1$ nunca será satisfeito. Visto que o problema da mochila 0-1 é o problema mochila disjunta com $E = \varnothing$, então ambos os problemas são NP-difíceis.

Também é possível relacionar outros problemas ao problema da mochila disjunta, como o *bin packing* com conflitos (JANSEN; ÖHRING, 1997), onde o mesmo conceito de que um dado par proibido não pode ser colocado na mesma *bin*. Da mesma forma, alguns problemas de agendamento (*scheduling*) (BODLAENDER; JANSEN, 1993) podem ser relacionados tanto ao *bin packing* quanto ao problema da mochila disjunta.

2.4.3 Mochila com penalidades

Também pode ser apresentada outra variação do problema da mochila similar ao problema da mochila disjunta, onde ao invés de considerar um par proibido, é permitido que este esteja na solução, desde que uma penalidade seja aplicada. Novamente, esta é uma versão ainda mais generalizada que o problema disjunto, carregando a mesma denominação de complexidade, sendo NP-difícil. A definição completa do problema da mochila com penalidades (CERULLI et al., 2020) pode ser novamente feita a partir das definições feitas para o problema da Mochila 0-1, adicionando a definição de um novo conjunto $F \subseteq X \times X$ de pares $(i,j)_k$ penalizados, cujo tamanho é $l = |F|$, e $i, j \in X$. Também é necessário definir um valor para tais penalidades, aqui denotados por d_k como sendo a penalidade do par $(i,j)_k$. Com isso, tem-se o seguinte modelo de programação inteira:

$$\max \quad \sum_{i \in X} p_i x_i - \sum_{k=1}^{l} v_k d_k \tag{8}$$

$$\text{s. t.} \quad \sum_{i \in X} w_i x_i \leq b \tag{9}$$

$$x_i + x_j - v_k \leq 1 \qquad \forall (i,j)_k \in F \tag{10}$$

$$x_i \in \{0,1\} \qquad \forall i \in X \tag{11}$$

$$0 \leq v_k \leq 1 \qquad \forall k \in \{1,...,l\} \tag{12}$$

As novas restrições (10) têm um papel similar às restrições análogas a elas no problema da Mochila Disjunta (6), mas, desta vez, quando $x_i = 1$ e $x_j = 1$, forçamos que $v_k = 1$, a qual é a única forma de a restrição ser satisfeita, "ativando" o acréscimo da respectiva penalidade na função objetivo (8).

Mesmo que seja um problema mais amplo que pode ser visto como uma relaxação das restrições do problema da mochila disjunta, pouco material pode ser encontrado especificamente sobre o problema da mochila com penalidades.

Para uma breve ilustração, pode ser considerado o seguinte exemplo:

- Existem 5 itens para serem escolhidos, com os seguintes atributos:

Item	Peso	Valor
1	3	22
2	13	5
3	9	16
4	18	23
5	7	24

- Existem 5 penalidades atribuídas:

Par	Penalidade
(1, 5)	2
(2, 3)	9
(2, 5)	14
(3, 4)	14
(4, 5)	4

O problema é selecionar um conjunto de itens cuja soma dos pesos não ultrapasse 18 e que este conjunto tenha a soma dos valores a maior possível. Ao considerar o problema da Mochila 0-1 (2.4.1), a solução é:

Item	Peso	Valor
1	3	22
5	7	24
	10	46

No entanto, se encararmos a penalidade como proibição, essa solução é invalida, e a solução para o problema da Mochila Disjunta (2.4.2) é:

Item	Peso	Valor
3	9	16
5	7	24
	16	40

Já considerando as penalidades de fato, o ganho é maior (e também é a mesma solução do problema que não considera essa restrição), e, sendo assim, um exemplo de que para alguns problemas serem penalizados, ao em vez de restringidos, pode levar a melhores soluções.

Item	Peso	Valor
1	3	22
5	7	24
	10	46

2.5 Métodos Exatos

A resolução de problemas de otimização pode se dar de diversas formas. O objetivo, de forma geral, é buscar a melhor solução. Existem diversos métodos de encontrarmos essa melhor solução, no entanto, deve-se deixar claro que essa tarefa pode ser custosa. Para o problema da mochila, por exemplo, por ser um problema de verificar permutações (verificar todos os possíveis subconjuntos de itens), sua complexidade é a de uma função fatorial, ou seja, quando o número de itens (n) aumenta linearmente, o número de verificações (e consequentemente o número de possíveis subconjuntos) cresce similarmente a um crescimento fatorial ($n!$).

Dentre as várias maneiras de se encontrar uma solução exata, podemos começar pela mais simples, uma busca enumerativa, ou seja, verificando todas as possíveis soluções. Por motivos já citados, esse método se torna inviável mesmo para um problema com entrada razoavelmente pequena (como um problema que considera permutações, por conta de sua complexidade fatorial) e a partir disso, ajustes podem ser feitos para que se tenha um método mais eficiente.

Dito isto, um método exato é aquele capaz de encontrar a melhor solução possível. Para este contexto de otimização combinatória, um método exato enumera sistematicamente

todas as soluções possível e descarta soluções individualmente ou em conjunto. Uma possível visualização de um método exato é considerar uma árvore formada por todas as possibilidades de caminhos percorridos por tal método entre soluções.

Problemas combinatórios são normalmente formulados como problemas de otimização inteira linear, em que podem ser empregadas estruturas geométricas, como hiperplanos e politopos, para reduzir o número de possíveis soluções a serem verificadas. Esta abordagem é consideravelmente mais complexa, no entanto, é construída para problemas mais gerais do que os aqui já apresentados.

Citando alguns dos métodos exatos mais conhecidos, temos o *Branch-and-Bound* (LAWLER; WOOD, 1966), onde certas propriedades e restrições impostas ao problema são usadas para eliminar, por exemplo, "galhos" da árvore de soluções. Ainda existem métodos que se utilizam de ideias mais geométricas, como métodos de corte do plano (MARCHAND et al., 2002), que pode ser traçado até um dos nomes mais conhecidos no estudo de otimização linear, Dantzig (DANTZIG, 1963). Ou ainda o *Branch-and-Cut* (MITCHELL, 2002), a qual é a união do anteriormente citado *Branch-and-Bound* e o método do plano de corte. É importante ressaltar que, por mais que estes métodos empreguem conceitos e mecanismos capazes de reduzir o tempo de execução, ainda são métodos exatos que certificam seus resultados por mostrar que a solução encontrada é melhor que todas as outras soluções possíveis. Mostrar isso em um problema NP-difícil (como o PMP, por exemplo) ainda é uma tarefa custosa. Portanto, para certos casos de uso, justifica-se o uso de métodos de menor tempo de execução cujo objetivo é pelo menos produzir uma solução "próxima" ao que seria o ótimo.

2.6 Heurísticas e Meta-heurísticas

Nem sempre existem recursos disponíveis para a utilização de métodos exatos, naturalmente por sua complexidade (considerando a inviabilidade temporal de resolução de grandes problemas NP-difíceis). A partir disso podemos prever métodos capazes de entregar soluções que, mesmo não sendo uma solução ótima, é próxima o suficiente para ser considerada viável. E é nesse ponto que temos heurísticas e meta-heurísticas (VOß, 2001).

Métodos heurísticos são aqueles que se utilizam de métodos mais inteligentes para a construção ou transformação de soluções a fim de se encontrar melhores soluções (KOKASH, 2005). Estudos sobre a aplicação de heurísticas cada vez mais refinadas podem ser encontrados, principalmente sobre problemas NP-difíceis. O resultado de um método heurístico muito provavelmente não será uma solução ótima, mas muito possivelmente uma boa aproximação, principalmente considerando cenários em que não se faz necessário ter todas as soluções, não é necessário encontrar a solução ótima, e em um tempo significativamente menor.

As chamadas meta-heurísticas (BLUM; ROLI, 2003) trabalham em um nível de abstração mais alto que as heurísticas, normalmente utilizando-se de alguma em sua estrutura. Uma possível definição para uma meta-heurística é um processo iterativo que controla uma ou mais heurísticas com a finalidade de produzir soluções de qualidades, transformando uma ou mais soluções completas, ou incompletas, em cada iteração (VOSS et al., 1998).

Um fator deve ser considerado enquanto analisando estas abordagens para a resolução do problema: o balanço entre diversificação e intensificação de soluções (YANG; DEB; FONG, 2014). A diversificação significa gerar diversas soluções de modo a preencher melhor todo o espaço de busca, ou seja, uma busca em escala global. Por outro lado, a intensificação consiste em gerar soluções apenas em uma região considerada promissora, a métrica considerada. Normalmente, em diversos métodos, a intensificação ocorre quando o

método seleciona uma região e tenta refinar a melhor solução presente nela, e, quando não é mais possível refinar (seja por tempo ou por exaustão), uma nova região é considerada, diversificando a solução.

2.6.1 Guloso e Carrossel Guloso

Os métodos baseados na heurística gulosa provavelmente são um dos mais simples de se conceber. Em seu conceito mais simples, um método guloso inicia com uma solução vazia e analisa o espaço de busca do problema, ou seja, o seu domínio, buscando o melhor elemento e adicionando à solução. Após a adição, o elemento também é removido do espaço de busca e o método volta-se à tarefa de analisar o espaço de busca restante. Esse processo iterativo ocorre até não existir mais elementos viáveis ou algum outro critério de parada. Um esboço para o método pode ser visto no Algorítimo 1.

Algoritmo 1: Método Guloso

```
1 Função Guloso(X):
2     s ← {∅}
3     while X ≠ ∅ ∧ ¬ parada(X, s) do
4         v ← melhor(X)      // Encontra o melhor elemento em X com algum critério pré-estabelecido
5         X ← X \ {v}
6         s ← s ∪ {v}
7     return s
8
```

Embora simples e muito útil, métodos gulosos, principalmente em grandes problemas, não tem bons resultados em relação à otimalidade. O espaço de busca visto por essa forma mais simples de método guloso é limitado, comparável com um único caminho da raiz ao nó da árvore atravessada por um método *branch-and-bound*. Ainda há outra característica problemática no método guloso, na qual a escolha de um elemento não permite que a solução final se aproxime da solução ótima no futuro, uma vez que adicionado um elemento, não se pode removê-lo da solução.

Pensando nisso, foi construída uma meta-heurística para tentar minimizar o impacto de escolhas iniciais de métodos gulosos. O método do Carrossel Guloso (CERRONE; CERULLI; GOLDEN, 2017) trata soluções parciais de um método guloso em um estilo temporal. O método inicialmente encontra uma solução gulosa completa S de tamanho $|S|$, descarta os últimos $\beta|S|$ elementos adicionados na solução, e executa $\alpha|S|$ iterações. Cada iteração remove o elemento mais antigo adicionado na solução e adiciona um elemento escolhido da mesma forma que no método guloso. Ao final, a solução é completada até que nenhum elemento possa ser adicionado, também usando o método guloso (Algoritmo 2).

2.6.2 *Iterated Local Search*

Para os problemas vistos, a transformação de uma solução em outra é simples, por exemplo, trocando um vértice de um grafo que esteja na solução por um que não esteja. Dessa forma, nota-se que a partir de uma solução, é possível definir uma vizinhança, e com isso pode perceber uma característica interessante. Como a otimalidade de uma solução é mensurável, existe um caso onde dada uma solução, nenhuma outra solução vizinha é melhor. Essa característica nos ajuda a definir um ótimo local.

Esse processo de transformar uma solução em outra até que se tenha um ótimo local é chamado de busca local. Este pode ser feito de diversas formas, basta que exista uma vizinhança bem definida. Por associação, observa-se que a existência de um ótimo local

Algoritmo 2: Carrossel Guloso

```
1  Função Carrossel(X, α, β):
       /* Supondo que a função Guloso retorna uma FIFO, com os elementos do mais antigo para o mais
          novo                                                                                  */
2      S ← Guloso(X)
3      X ← X \ S
4      t ← |S|
5      S ← S[0..βt]
6      for i ← 0 to αt do
7          v ← pop_front(S)
8          X ← X∪ {v}
9          u ← melhor(X)
10         push_back(S, u)
11     while X ≠ ∅ ∧ ¬ parada(X, S) do
12         v ← melhor(X)
13         X ← X \ {v}
14         S ← S ∪ {v}
15     return S
16
```

implica na existência de um chamado ótimo global, a qual é a melhor solução dentre todas as soluções. A existência de um método de busca local eficiente auxilia a reduzir o espaço de busca.

Pensando nisso, é possível pensar em um método que busque entre os ótimos locais. Esse método é o já conhecido *Iterated Local Search* (ILS) (LOURENçO; MARTIN; STüTZLE, 2018). O ILS, de forma breve, consiste em um processo iterativo, onde a partir de uma solução já existente é aplicada uma perturbação para mover a solução para uma vizinhança com um ótimo local diferente. Em seguida, é aplicado uma busca local para se chegar a esse ótimo local de fato e considerando esse novo ótimo local como global por meio de algum critério de aceitação. O controle da intensificação também pode ser feito, a qual é a busca nas proximidades de ótimos locais, e a diversificação, a qual é a busca em outras vizinhanças por ótimos locais. Esse controle é normalmente feito a partir de trajeto do processo iterativo. Uma estrutura básica para o ILS é apresentada no Algoritmo 3.

Algoritmo 3: ILS

```
1  Função ILS(X):
2      s ← solucao_inicial(X)
3      s* ← busca_local(s)
4      while Criterio de para não atingido do
5          s' ← pertubação(s*)
6          s' ← busca_local(s')
7          s* ← aceitação(s*, s', histórico)
8      return s*
9
```

2.6.3 *Variable Neighborhood Descent*

Como dito anteriormente, soluções possuem vizinhanças, e a partir dessas vizinhanças é possível definir o que seria um ótimo local. No entanto, é necessário definir sobre quais vizinhanças estamos trabalhando, visto que um ótimo é local considerando uma certa vizinhança (ou estrutura de vizinhança), pode ser que não seja mais o ótimo ao

considerar uma outra estrutura. A partir disso, podemos refinar a ideia do máximo global como sendo o melhor ótimo local considerando todas as vizinhanças possíveis. Portanto, uma boa abordagem para uma busca local é verificar diferentes estruturas de vizinhança.

Devido a isso, o método *Variable Neighborhood Search* (VNS) (HANSEN et al., 2018) pode ser concebido. Este método de busca local consiste justamente em verificar as vizinhanças pré-determinadas de uma dada solução até que se encontre um ótimo que seja o ótimo local em todas as possíveis vizinhanças. Uma possível notação para uma estrutura de vizinhança de uma solução S é $N_k(S), k = 1, ..., k_{max}$, onde k_{max} é o total de estruturas de vizinhança que estamos analisando. Dado isto, ainda é necessária uma forma de alterar a vizinhança analisada. Ao considerar uma forma determinística de variar as estruturas de vizinhança tem-se o VND, *Variable Neighborhood Descent*. O algoritmo que visa maximizar uma função f, a partir de uma solução inicial S e utilizando-se de k_{max} estruturas de vizinhanças, pode ser visto no Algoritmo 4.

Algoritmo 4: VND

```
1  Função VND(f, S, k_max):
2      k ← 1
3      while k < k_max do
4          S' ← melhor(N_k(S))
5          if valor(S') > valor(S) then
6              S ← S'
7              k ← 1
8          else
9              k ← k + 1
10     return S
11
```

Ainda é necessário definir como buscar o melhor de uma vizinhança. Comumente são utilizados dois métodos de busca, o *best improvement* e *first improvement*. O método *best improvement* busca realmente a melhor solução de uma vizinhança, analisando todas as opções e entregando a melhor. Já o método *first improvement* troca a melhor solução pelo ganho temporal. O *first improvement* analisa uma solução da vizinhança por vez, e ao encontrar uma solução que seja melhor que a atual, essa é entregue como a melhor da vizinhança.

3 Revisão da Literatura

Este capítulo é dedicado à apresentação e discussão de trabalhos relacionados ao tema pertinente. Mais precisamente, serão vistos trabalhos sobre resoluções para o problema do conjunto máximo independente generalizado, dada sua íntima relação ao PMP, resoluções do problema da mochila, e, por fim, sobre o mesmo problema com a adição de mais restrições.

3.1 Métodos para problemas de conjuntos independentes generalizados

Nesta seção, são apresentados métodos para a resolução do problema do conjunto independente máximo generalizado (GISP) (COLOMBI; MANSINI; SAVELSBERGH, 2017). Por mais que o problema abranja diversos outros por ser uma generalização, não se encontra trabalhos especificamente sobre o problema com facilidade.

Uma aparição razoavelmente antiga de um problema GISP foi no problema da seleção de partições de floresta para colheita (HOCHBAUM; PATHRIA, 1997). Cada partição tem duas decisões envolvidas, a primeira tendo relação com o valor da colheita da partição e penalidades sobre a colheita de partições adjacentes, e a segunda decisão, a respeito do valor do impacto ecossistêmico, considerando a criação das bordas entre as partições. Ambos os problemas foram traduzidos para o GISP. Em seguida, métodos de resolução utilizando para o GISP em determinados tipos de grafos (bipartites) foram desenvolvidos, inclusive, com tempo de execução polinomial. Além de uma introdução à definição do que é um GISP, como resultado, o trabalho contribuiu para a expansão da área de estudo, propondo alternativas aos métodos de programação linear e dinâmica.

Para relacionar o GISP e o PMP, é possível imaginar uma instância do PMP em que podemos desconsiderar o peso de todos os itens, fazendo com que o problema se torne encontrar uma solução ótima que não possua pares proibidos. Considerando-se cada item um vértice de um grafo e cada par proibido uma aresta entre seus respectivos vértices, temos o problema do conjunto independente. O problema do conjunto independente generalizado, além das arestas proibidas, considera a existência de arestas que podem ser admitidas na solução, desde que se pague um custo. Logo, podemos transformar uma instância do problema do conjunto independente generalizado, em que todas as arestas são permitidas com custos, em uma instância do PMP, ao atribuirmos um novo atributo de peso à cada vértice. O inverso também pode ser feito quando os pesos dos itens forem desconsideráveis.

Por fim, Nogueira e Pinheiro (NOGUEIRA; PINHEIRO; TAVARES, 2020) propuseram a aplicação da meta-heurística ILS ao GISP. O método desenvolvido tem como busca local o VND equipado com duas estruturas de vizinhança. A primeira adiciona um vértice na solução, mas antes se é feita uma escolha, remover vértices adjacentes ao adicionado ou remover arestas removíveis entre o vértice adicionado e outros na solução, sendo escolhido o que causar uma melhora no valor da solução. A segunda estrutura de vizinhança adiciona dois vértices, seguindo as mesmas regras que a outra estrutura de vizinhança. Além disso, estruturas de dados e arranjos ótimos das informações do problema foram desenvolvidos para reduzir o tempo de avaliação e operações durante o processo. Foram feitas comparações entre este e outros métodos do estado da arte para o problema, e como resultado, o método foi mais eficiente que os demais tanto em tempo de execução quanto em qualidade da solução.

3.2 Métodos para problemas da mochila 0-1

Esta seção é dedicada às resoluções do problema da mochila em sua forma mais simples. Por mais que seja um tópico incansavelmente discutido, não se pode deixar de lado, uma vez que a partir dos métodos para este problema mais simples, podemos derivar ou tomar ideias para a resolução de métodos mais complexos.

Um dos trabalhos clássicos sobre o problema da mochila 0-1 foi o de Peter Kolesar (KOLESAR, 1967). Seu trabalho propõe um método *branch-and-bound* em que se é construído nós representando classes de soluções cada vez menores a cada *branch*, até que se tenham acabado as possibilidades de gerar nós, seja pelo fato de que o nó só representa uma única solução, ou tenha descumprido o *upper bound*, um limitante para a otimalidade da solução. *Upper bound* este que é recalculado a cada geração de nó. Seus resultados foram impactados pela tecnologia da época, mas mesmo assim foi considerado um bom resultado, principalmente pelo fato de que quase todas as instâncias de problemas foram resolvidas de forma ótima.

O trabalho de Toth (TOTH, 1980) foi voltado para o desenvolvimento de novos métodos de programação dinâmica (BELLMAN, 1966) para o problema da mochila 0-1. O seu método consiste em resolver subproblemas, com quantidade de itens e capacidade reduzida. Assim, para um problema com n itens e capacidade b, o método é derivado de uma função recursiva, $f_m(z)$, que define um subproblema onde são considerados apenas os m primeiros item e uma capacidade z ($0 \leq z \leq b$):

$$f_m(z) = \max \left\{ \sum_{i=1}^{m} p_i x_i \mid \sum_{i=1}^{m} w_i x_i \leq z \right\}.$$

De onde se pode derivar:

$$f_m(z) = \begin{cases} f_{m-1}(z), & \text{se } 0 \leq z < w_m \\ \max(f_{m-1}(z), f_{m-1}(z - w_m) + p_m), & \text{se } w_m \leq z \leq b \end{cases}$$

Com os seguintes casos-base:

$$f_1(z) = \begin{cases} 0, & \text{se } 0 \leq z < w_1 \\ p_1, & \text{se } w_1 \leq z \leq b \end{cases}$$

Em que a solução para a instância original é dado por achar o valor de $f_n(b)$.

O método para a solução dessa recursão é equipado com técnicas capazes de reduzir significantemente o custo computacional, independente do tamanho da instância do problema. Essas técnicas tanto reduzem o número de estados presentes na execução do método, quando estabelecem fronteiras facilitando na busca da solução. Seus resultados mostram que o método é robusto, principalmente quando comparado a métodos *branch-and-bound* em grandes instâncias.

Outro estudo, desta vez do início do milênio (MARTELLO; PISINGER; TOTH, 2000), tem como foco explorar e comparar o que se tem de recente para o problema da mochila 0-1. Martello, Pisinger e Toth compilaram métodos *branch-and-bound* básicos, métodos que se utilizam do conhecimento das soluções parciais mais interessantes, métodos de programação dinâmica e métodos que trabalham em cima de limitantes mais bem estabelecidos. Como experimento, os autores utilizaram os métodos (incluindo outro que combina duas técnicas) para resolver um certo conjunto de instâncias de diferentes características. Podemos considerar que a maior contribuição desse trabalho foi indicar qual o método mais eficiente para determinados tipos de instâncias.

Há também um método guloso proposto por George B. Dantzig (DANTZIG, 1957). Este método, sem perda de generalidade, assume que $p_i/w_i \geq ... \geq p_n/n$, isto é, os itens são ordenados pela razão peso-valor. Dessa forma, o método guloso funciona por coletar um item por vez, em sequência, do item com maior razão peso-valor para o menor, até que se esgote a capacidade da mochila, ou se tenha incluído todos os itens. Este método, por mais facilmente aplicável que seja, pode levar a soluções longe do ótimo com facilidade, quando, por exemplo, a inclusão de um item inibe a inclusão de outros cuja soma de valores é maior e soma de pesos é menor que o do item já incluído em questão. Existem também estudos que consideram soluções ótimas de instâncias parciais (normalmente pequenas em tamanho), isto é, com itens faltando. Esses estudos apresentam métodos que re-otimiza as soluções encontradas para instâncias "incompletas" quando os itens que faltam são adicionados (ARCHETTI; BERTAZZI; SPERANZA, 2010), onde, novamente, foram usados métodos exatos e gulosos.

A hibridização de métodos também é algo recorrente na literatura, como o uso do *simulated annealing* (NAHAR; SAHNI; SHRAGOWITZ, 1986) (método que simula o processo de formação de uma estrutura cristalina) em conjunto com algoritmos genéticos (LIN; KAO; HSU, 1993), de forma que o uso desses algoritmos genéticos acelerem a execução e contornem algumas das dificuldades do *simulated annealing*.

Métodos não tão facilmente relacionados ao problema da Mochila 0-1 também foram aplicados, como um método baseado em colonias de formigas (SHI, 2006) onde foram consideradas o uso de estratégias para o problema do Caixeiro Viajante (LIN, 1965), outro problema combinatório famoso. Outra forma não muito convencional para o problema é o uso de uma busca simplificada de harmonia binaria (KONG et al., 2015), uma técnica inspirada em habilidades de improvisação de músicos. Ainda podemos citar métodos baseados em exames de partículas (GHERBOUDJ; CHIKHI, 2011), como, por exemplo, otimização da borboleta-monarca (FENG et al., 2017), onde se é usado o conceito de migração dos indivíduos. Temos também um método inspirado em características sociais de baleias (ABDEL-BASSET; EL-SHAHAT; SANGAIAH, 2019), que se mostrou superior em qualidade da solução quando comparado aos métodos considerados estado-da-arte na época.

3.3 Métodos para a mochila disjunta

O Problema da Mochila Disjunta, é mais reconhecido que o Problema da Mochila com Penalidades e possui uma grande variedade de métodos a seu respeito, muito disso é devido à sua relação com problemas muito bem conhecidos na literatura, cliques e conjuntos máximos independentes (Seção 2.3.1).

Dentre estes métodos, temos métodos evolutivos como o *Scatter Search* (HIFI; OTMANI, 2012), métodos gulosos (YAMADA; KATAOKAT; WATANABET, 2002), métodos derivados de aplicações em grafos (PFERSCHY; SCHAUER, 2016), *augmented rounding algorithms* (HIFI; MOUNIR, 2009), buscas locais reativas (HIFI; MICHRAFY, 2006), e diversos muitos outros métodos exatos, como, por exemplo, métodos *branch-and-bound* (BETTINELLI; CACCHIANI; MALAGUTI, 2017), métodos de programação dinâmica (GURSKI; REHS, 2019) (os quais são métodos usuais para a família de problemas da mochila), entre outros (SALEM et al., 2018; CESELLI; RIGHINI, 2006b; CROCE; PFERSCHY; SCATAMACCHIA, 2019; HIFI; MICHRAFY, 2007).

3.4 Métodos para o Problema da Mochila com Penalidades

Para o problema objeto deste trabalho, o Problema da Mochila com Penalidades (PMP), diversas aplicações podem ser citadas, como já feito, principalmente em casos onde res-

trições de proibição podem levar a soluções piores na prática. De certa forma, a maioria dos métodos de resolução para os outros problemas da mochila podem ser adaptados para esse.

No geral, o PMP ainda não foi bem explorado até o momento na literatura. Considerando isso, e ainda o desenvolvimento da meta-heurística do Carrossel Guloso (CERRONE; CERULLI; GOLDEN, 2017), Cerulli, D'Ambrosio, Raiconi e Vitale (CERULLI et al., 2020) definem e tratam o problema. Neste caso, o problema foi resolvido usando o Carrossel Guloso, fazendo o uso da heurística gulosa mais usual para o problema (avaliando a razão entre valor e peso). O estudo foi demonstrativo e comparativo com o método exato usado pela biblioteca CPLEX (ILOG, 2009). O resultado obtido, para os autores, foi satisfatório, encontrando soluções razoavelmente próximas ao ótimo em tempo hábil.

Em seguida, o Carrossel Guloso foi usado em conjunto com Algoritmos Genéticos (BEASLEY; BULL; MARTIN, 1993) novamente para o PMP (CAPOBIANCO et al., 2021), concebendo o método Genetic Algorithm-Carousel Greedy (GA-CG). Segundo os autores, esse novo método quando comparado aos outros dois, também testados por eles, mostrou resultados melhores em todas as instâncias aplicadas. Por fim, e mais recentemente, um novo método, MemeticFS (MA) que estende o GA-CG (D'AMBROSIO et al., 2023) trouxe ainda melhores resultados, mesmo endereçando uma generalização do problema.

Está subseção trata sobre os dois métodos mais recentes, o GA-CG e MA, que incorporaram o desenvolvimento e ideias dos métodos anteriores a eles, uma vez que todos, em algum nível, aplicam o método guloso.

3.4.1 GA-CG

O GA-CG (Genetic Algorithm-Carousel Greedy) (CAPOBIANCO et al., 2021) é apresentado como uma meta-heurística hibrida que junta as vantagens de algoritmos genéticos e o Carrossel Guloso (Carousel Greedy), onde cada solução vem de um processo evolucionário tipico de algoritmos genéticos.

Antes de abordar diretamente o método, é preciso visitar a metodologia gulosa aplicada a ele. O método guloso utilizado (CERULLI et al., 2020) pode ser visto como uma extensão da heurística comumente usada no Problema da Mochila 0-1 para o PMP. A heurística consiste em simplesmente adicionar um item i à solução S por vez, até que não se tenha mais candidatos e que i tenha a maior razão valor por peso dentre os candidatos. O valor do item i é simplesmente p_i quando não existe outro item j na solução que seja um par junto à i. Quando existe um ou mais itens j que façam pares, o valor é $p_i - \sum d_k$, onde d_k é a penalidade de cada par. Sendo assim, o valor de i é de fato o valor a ser agregado na solução, já considerando as penalizações trazidas por sua inclusão.

O Carrossel Guloso (CERULLI et al., 2020) faz uso da mesma mecânica de seleção que o método guloso usual adaptado ao problema, já citado anteriormente, sendo executado até que não se tenha mais candidatos, resultando em uma solução inicial para o método. Se fez necessário também funções auxiliares, uma para remoção de uma porcentagem (β) dos últimos itens (mais recentemente adicionados) da solução, outra que retorne o melhor item de um conjunto usando a heurística gulosa (`melhor`). O pseudocódigo do Carrossel Guloso pode ser visto no Algoritmo 5.

Por fim, o GA-CG (CAPOBIANCO et al., 2021) segue as estruturas usuais do que seria um algoritmo genético, com o incremento do uso do Carrossel Guloso. Inicialmente, o método gera uma população inicial aleatória, onde cada indivíduo dessa população é uma solução do problema, e, em seguida, seleciona o melhor individuo. Seguindo dai se inicia a fase iterativa, em que cada iteração consiste em escolher um par de soluções, fazer um crossover em ambos, aplicar uma mutação no resultado do crossover, aplicar o Carrossel Guloso (ignorando o processo de geração de solução inicial) usando o individuo resultante

Algoritmo 5: Carrossel Guloso

```
1  Function Carrossel Guloso(P):
2      S ← Guloso(P)
3      S' ← remove_ultimos(S, β)
4      t ← |S'|
5      for i ← 1 to αt do
           /* Considerando que S' seja uma FIFO                    */
6          pop_front(S')
7          Candidatos ← {z, ∀z ∈ S', w_z ≤ capacidade_restante(S')}
8          i* ← melhor(Candidatos)
9          push_back(S', i*)
10     while Existem candidatos válidos do
11         i* ← melhor(Candidatos)
12         S' ← S' ∪ {i*}
13     return S'
14
```

Algoritmo 6: GA-CG

```
1  Function GA-CG(P):
2      Pop ← população_aleatoria(P)
3      C* ← melhor(Pop)
4      for i ← 1 to NumIters do
5          C1, C2 ← escolhe_par(Pop)
6          C3 ← escolhe_excluso(Pop)
7          C ← crossover(C1, C2)
8          C ← mutação(C)
9          C ← CGInicializado(C)
10         if C ∉ Pop then
11             Pop ← Pop \ {C3} ∪ {C}
12             C* ← melhor(Pop)
13     return C*
14
```

da mutação (incrementando o valor da solução até o máximo possível). Em seguida é selecionado um indivíduo candidato à exclusão da população, e, por fim, caso o resultado do Carrossel Guloso não esteja na população, substituímos o candidato à exclusão com o novo individuo, e procuramos o novo melhor individuo da população. O pseudocódigo é apresentado no Algoritmo 6.

3.4.2 MA

É possível imaginar uma generalização para o PMP, bastando fazer apenas uma consideração a mais: em vez de aplicar penalidades somente a pares de itens quando ambos estão presentes na solução, uma penalidade pode estar associada à qualquer subconjunto de itens de tamanho arbitrário que esteja presente na solução. Assim, faz-se com que o PMP seja um caso especial em que todo subconjunto é de tamanho 2. Esse novo problema mais abrangente foi chamada do Problema da Mochila com Conjuntos Penalizados (D'AMBROSIO et al., 2023). Junto a esse problema também foi apresentado o método MemeticFS, ou apenas MA (Algoritmo 7), que foi concebido como uma extensão do GA-CG admitindo tais subconjuntos penalizados. Este método, além dos ajustes em cima do GA-CG para considerar mais do que pares, também considera em seu processo iterativo

a aplicação de uma busca local usando três operações simples, adição, remoção e troca.

As três operações executam exaustivamente em seus respectivos espaços de busca. A operação de adição apenas busca um item que quando adicionado melhora a solução. A operação de remoção busca um item que quando removido, melhora a solução. E, por fim, a operação de troca que verifica todas os pares possíveis entre itens dentro e fora da solução, no qual um será removido e outro será adicionado, e seleciona o melhor par (o par que melhora a solução após feita a troca). A busca local executa os operadores na ordem que foram apresentados.

Algoritmo 7: MA

```
1  Function MA(P):
2      Pop ← população_aleatoria(P)
3      C* ← melhor(Pop)
4      it_no_impr ← 0
5      for i ← 1 to NumIters do
6          C1, C2 ← escolhe_par(Pop)
7          C3 ← escolhe_excluso(Pop)
8          C ← crossover(C1, C2)
9          C ← CGInicializado(C)
10         C ← busca_local(C)
11         if valor(C) ≥ valor(C*) then
12             C* ← C
13             it_no_impr ← 0
14         else
15             it_no_impr ← it_no_impr + 1;
16         if it_no_impr > max_it_impr then
17             return C*
18         if C ∉ Pop then
19             Pop ← Pop \ {C3} ∪ {C}
20     return C*
21
```

Além do que já foi comentado, também podemos ver pequenos ajustes feitos em relação ao GA-CG, como o método parar após um certo número de iterações sem melhoria (linhas 13–17), e uma atualização da melhor solução encontrada mais inteligente (linhas 11–12).

4 ILS-VND para o problema da Mochila com Penalidades

Este capítulo é destinado à descrição da metodologia aplicada à construção do método para a resolução do problema da mochila com penalidades.

4.1 Algoritmo

Esta seção apresenta o algoritmo ILS-VND proposto. O ILS é uma técnica de otimização bem conhecida que começa com uma solução inicial e, em seguida, a refina iterativamente por meio de procedimentos de perturbação e busca local. A busca local é usada para intensificar a busca em áreas promissoras, enquanto a perturbação é aplicada para diversificar a busca e escapar potencialmente de ótimos locais. Em nosso método, a busca local é realizada usando uma abordagem VND.

O pseudocódigo para o ILS-VND proposto pode ser visto no Algoritmo 8 e está dividido em quatro partes principais: a inicialização (linhas 2–6), a perturbação (linha 9), a busca local (linhas 3 e 10) e o critério de aceitação (linha 11). Avaliamos a solução usando a função `valor`, que nada mais é que a função objetivo (8).

Nas subseções seguintes, essas partes são detalhadas. Após a inicialização de uma solução usando uma função que implementa uma heurística gulosa `solução_inicial(P)` (linha 2), nosso algoritmo utiliza a função `busca_local` para refiná-la (linha 3). Em seguida, o ILS-VND inicializa algumas variáveis usadas na função `aceitação` (linhas 4–6). No loop principal (linhas 7–11), o algoritmo cria uma cópia local S' da solução atual S (linha 8), perturba-a (linha 9) e aplica a busca local a essa nova solução (linha 10). Na linha 11, o ILS-VND usa a função `aceitação` para testar se a nova solução S' será mantida como a nova solução atual S.

Algoritmo 8: ILS-VND. Recebe como entrada uma instância KPF, representada como P, que inclui todas as informações necessárias: o conjunto de itens X, os pesos dos itens w, os valores dos itens p, a capacidade máxima de peso da mochila b e o conjunto de penalidades F. Retorna a melhor solução $S^* \subseteq X$ encontrada durante a sua execução. c_1 juntamente com outras constantes usadas em `aceitação` estão fixadas no código e independem das instâncias.

```
1  Function ILS-VND(P := {X, w, p, b, F }):
2      S ← solução_inicial(P ) // Seção 4.1.1
3      S ← busca_local(S)
4      S* ← S
5      k ← 1
6      melhor_local ← valor(S*)
7      while Condição de parada não alcançada do
8          S' ← S
9          S' ← perturbação(S', c1) // Seção 4.1.3
10         S' ← busca_local(S') // Seção 4.1.2
11         k, S, S*, melhor_local ← aceitação(S', S, k, S*, melhor_local) // Seção 4.1.4
12     return S*
```

4.1.1 Inicialização

A solução inicial S é gerada usando a função `solução_inicial(P)` (linha 2 do Algoritmo 8), que utiliza uma abordagem gulosa conhecida (CERULLI et al., 2020), a qual segue os moldes do que foi descrito na Seção 2.6.1. Essa abordagem seleciona o item mais valioso a cada iteração, levando em consideração também os custos de penalização associados aos itens correlacionados.

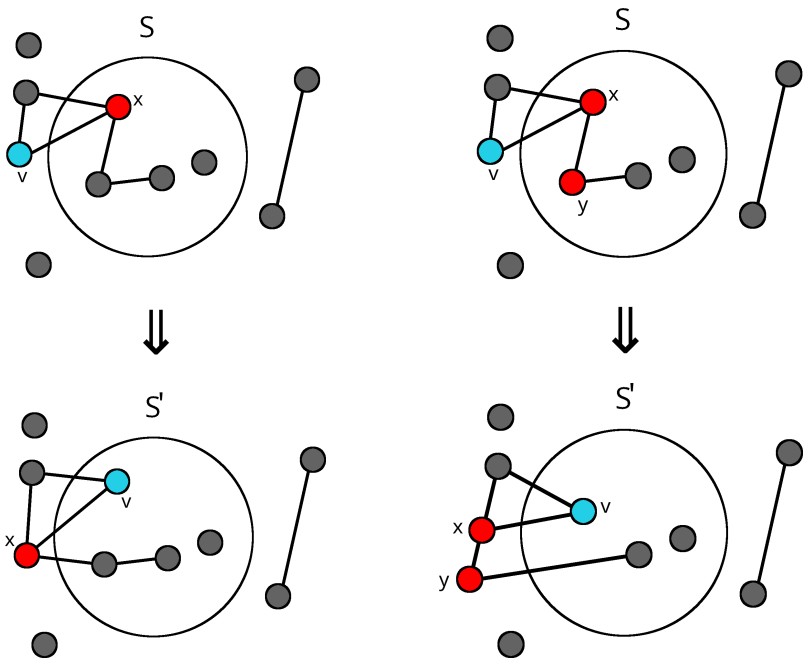
Especificamente, o valor p'_i de um item i é o seu valor original p_i quando não existe outro item j na solução que i faça par. Quando j faz parte da solução, p'_i é reduzido pelo valor da penalidade envolvida:

$$p'_i = p_i - \sum_{j \in S} d_k$$

onde d_k é a penalidade associada ao par $(i, j)_k$. Considerando os itens e penalidades indicadas na Seção 2.4.3 e tomando como exemplo uma mochila com itens 3 e 5, deseja-se adicionar o item 2. O item 2 possui penalidade com ambos os itens, logo, o valor que o item 2 tem de fato é $p'_i = 5 - d_{2,3} - d_{2,5} = 5 - 9 - 14 = -18$, indicando que adicionar o item i piora a solução. Logo, no contexto da inicialização, esse cenário nunca seria considerado, visto a natureza gulosa do método.

4.1.2 Busca Local

No algoritmo proposto, a busca local é usada após a inicialização de uma solução (linha 3) e após uma perturbação (linha 10). O procedimento de busca local utiliza uma estratégia VND, a qual é um método que explora de forma determinística várias estruturas de vizinhança aplicando operadores predefinidos (HANSEN et al., 2018). O VND utilizado aplica quatro estruturas de vizinhança: `swap-01`, `swap-10`, `swap-11` (Figura 1a) e `swap-21` (Figura 1b) para explorar o espaço de soluções.



(a) `swap-11` (b) `swap-21`

Figura 1: Swap de troca

Dentro de cada estrutura de vizinhança, um operador dedicado foi desenvolvido para selecionar uma solução que melhore o resultado. Para definir formalmente essas distintas estruturas de vizinhança, introduzindo três conjuntos auxiliares:

$$Z^S = \left\{ z \in X \setminus S \;\middle|\; \sum_{i \in S} w_i + w_z \leq b \right\}$$
$$Z^S_{x \in S} = \left\{ z \in X \setminus S \;\middle|\; \sum_{i \in S \setminus \{x\}} w_i + w_z \leq b \right\}$$
$$Z^S_{x,y \in S} = \left\{ z \in X \setminus S \;\middle|\; \sum_{i \in S \setminus \{x,y\}} w_i + w_z \leq b \right\}.$$

O conjunto Z^S inclui itens z que podem ser adicionados à solução S, enquanto Z^S_x contém itens z que podem ser adicionados se x for removido de S, e $Z^S_{x,y}$ inclui itens z que podem ser adicionados se tanto x quanto y forem removidos de S. É importante enfatizar que o elemento a ser adicionado é diferente dos que estão sendo removidos. A definição dos nossos quatro operadores envolve a seleção de uma solução a partir dos seguintes conjuntos:

$$\mathcal{N}_{01}(S) = \{S' : S' = S \cup \{z\}, z \in Z^S, \texttt{valor}(S') > \texttt{valor}(S)\}$$
$$\mathcal{N}_{10}(S) = \{S' : S' = S \setminus \{x\}, x \in S, \texttt{valor}(S') > \texttt{valor}(S)\}$$
$$\mathcal{N}_{11}(S) = \{S' : S' = (S \setminus \{x\}) \cup \{z\}, x \in S, z \in Z^S_x, \texttt{valor}(S') > \texttt{valor}(S)\}$$
$$\mathcal{N}_{21}(S) = \{S' : S' = (S \setminus \{x,y\}) \cup \{z\}, x,y \in S, z \in Z^S_{x,y}, \texttt{valor}(S') > \texttt{valor}(S)\}.$$

O Algoritmo 9 apresenta o procedimento de busca local proposto. Ele é dividido em duas fases. Na fase inicial (linhas 2-6), nosso algoritmo explora sequencialmente as vizinhanças `swap-11` e `swap-21`, enquanto na segunda fase (linhas 7-11), ele explora as vizinhanças `swap-01` e `swap-10`. Essa sequência foi escolhida pelo fato de que `swap-01` e `swap-10` pode ser considerados "ajustes finos", fazendo melhoramentos de apenas um item, os quais `swap-11` e `swap-21` não são capazes de fazer. A exploração de uma vizinhança é feita de maneira que a primeira solução melhorada encontrada seja adotada (*first improvement*), substituindo a solução atual. Foram desenvolvidas diversas estratégias para aumentar a eficiência da exploração das vizinhanças, como o uso de uma estrutura de dados de *heap* e uma lista tabu. Essas estratégias são detalhadas na Seção 4.2.

Algoritmo 9: Busca Local. Recebe como entrada uma solução para o KPF que contém todas as informações necessárias e retorna uma nova solução S, caso alguma das trocas tenha sido bem-sucedida, ou a mesma solução de entrada, caso contrário.

```
1  Function local_search(S):
2      while N_11(S) ≠ ∅ and N_21(S) ≠ ∅ do
3          if N_11(S) ≠ ∅ then
4              S ← S', de forma que S' ∈ N_11(S)
5          else if N_21(S) ≠ ∅ then
6              S ← S', de forma que S' ∈ N_21(S)
7      while N_01(S) ≠ ∅ and N_10(S) ≠ ∅ do
8          if N_01(S) ≠ ∅ then
9              S ← S', de forma que S' ∈ N_01(S)
10         else if N_10(S) ≠ ∅ then
11             S ← S', de forma que S' ∈ N_10(S)
12     return S
```

Com este algoritmo e conjuntos definidos, sabe-se quais os itens que serão avaliados. Entretanto, ainda é necessário definir uma sequência para a avaliação destes itens, ou seja, de que forma os itens x, y e z são selecionados em sequência para os conjuntos $\mathcal{N}_{01}, \mathcal{N}_{10}, \mathcal{N}_{11}$ e $\mathcal{N}_{21}$, principalmente pelo fato que se foi considerada uma busca *first improvement*. Com exceção do `swap-21`, todos os operadores iteram sobre os itens na ordem com que eles estão dentro do *array* que representa a solução (Seção 4.2). O `swap-21`, por ser mais complexo, considera a ordem com que os itens estão em determinadas estruturas de dados que serão apresentadas mais a frente.

4.1.3 Perturbação

A função de perturbação (linha 9 do Algoritmo 8) é parametrizada por k, que determina o número de itens a serem adicionados à solução. Especificamente, a função de perturbação adiciona k itens à solução, mesmo que isso exceda a capacidade da mochila. Para restaurar a viabilidade, essa função remove aleatoriamente itens que estavam na solução antes da perturbação até que a restrição de capacidade da mochila seja satisfeita.

Essa estratégia de perturbação permite que o algoritmo escape de ótimos locais e explore novas regiões do espaço de busca. O parâmetro k controla a intensidade da perturbação e pode ser ajustado para equilibrar as fases de intensificação e diversificação do algoritmo.

4.1.4 Critério de aceitação

A função `aceitação` na linha 11 do Algoritmo 8 é descrita no Algoritmo 10. Este algoritmo segue uma estrutura semelhante àquela delineada em (NOGUEIRA; PINHEIRO; SUBRAMANIAN, 2017). Uma solução candidata S' é sempre aceita como a nova solução atual se tiver um valor melhor do que a solução atual S (linha 4 do Algoritmo 10). O algoritmo também aceita soluções piores que S dependendo do valor do contador local k. Essa aceitação é feita indiscriminadamente, desde que sejam de fato soluções viáveis. Sempre que o contador k atinge um valor maior do que $|S|/2$, o algoritmo aplica uma operação de perturbação (usando um parâmetro diferente do usado nas iterações do ILS) na solução atual S (linha 15) e aceita a solução perturbada como a nova solução atual. Ao aceitar soluções que não melhoram, o algoritmo evita ficar preso em ótimos locais, facilitando assim a busca pelo ótimo global. O valor para essa condição ($|S|/2$) foi esco-

lhido por ser proporcional ao tamanho da solução em questão, uma vez que quanto maior a solução, justifica-se manter a busca nas mesma região em que se está no momento, dada o maior número de soluções vizinhas.

O valor de k é sempre redefinido para 1 quando uma nova solução é aceita (linhas 3 e 16). Além disso, ele é decrementado em duas condições: quando a melhor solução local é melhorada (linha 6) ou quando a melhor solução S^* é melhorada (linha 10). Por outro lado, se S não melhorar, o valor de k é incrementado (linha 12).

Finalmente, as constantes c_2, c_3 e c_4 (calibradas conforme descrito na Seção 5.3) desempenham um papel crucial no equilíbrio entre os aspectos de intensificação e diversificação do algoritmo. Vale a pena notar que o parâmetro usado na perturbação nas iterações do Algoritmo 3 difere do usado no Algoritmo 10. Isso se deve ao fato de que quando o contator k indica que se excedeu o número de iterações buscando naquela mesma região, a pertubação deve ser feita com um parâmetro diferente para iniciar a busca em uma região diferente (ou pelo menos uma nova solução com um número de itens diferente dos considerados em outras iterações).

Algoritmo 10: Critério de Aceitação. Recebe como entrada a solução da iteração atual S', a última solução aceita S, a variável de controle de aceitação k, a melhor solução global S^* e a variável *melhor_local* que armazena o valor antes da perturbação da última solução aceita. Retorna valores atualizados para k, S, S^* e *melhor_local*.

```
1  Function aceitação(S', S, k, S*, melhor_local):
2      if valor(S') > valor(S) then
3          k ← 1
4          S ← S'
5          if melhor_local < valor(S) then
6              k ← k − |S| / c2
7              melhor_local ← valor(S)
8          if valor(S*) < valor(S) then
9              S* ← S
10             k ← k − |S| * c3
11     else if k ≤ |S| / 2 then
12         k ← k + 1
13     else
14         melhor_local ← valor(S)
15         S ← perturbação(S, c4)
16         k ← 1
17     return k, S, S*, melhor_local
```

4.2 Exploração de vizinhanças

Como dito anteriormente, o método proposto utiliza quatro estruturas de vizinhança diferentes para percorrer o espaço de soluções do PMP. Podemos perceber que explorar essas vizinhanças pode aumentar consideravelmente o custo computacional, especialmente à medida que o tamanho do problema (número de itens ou pares) aumenta. Para contrabalancear o impacto dessas complexidades na eficiência do método, foram implementados mecanismos adicionais. Esses mecanismos foram projetados especificamente para aprimorar a seleção e avaliação de itens durante a execução, fazendo com que o algoritmo encontre um equilíbrio entre precisão e eficiência computacional. A seguir serão vistas explicações detalhada desses mecanismos. Primeiro, introduziremos as estruturas de dados

adotadas para esse fim e, em seguida, apresentaremos como essas estruturas de dados são usadas para uma exploração rápida das vizinhanças.

4.2.1 Estruturas de dados

Para uma exploração rápida das vizinhanças, nosso algoritmo utiliza as seguintes estruturas de dados: (i) três *arrays* com tamanho igual ao número de itens no problema, chamados de *array* de solução, *array* de valores e lista tabu, e (ii) duas estruturas de dados de *heap*. Todas essas estruturas de dados são atualizadas apenas durante a inclusão ou remoção de itens da solução.

***Array* de Solução**: Este *array* é dividido em dois segmentos. O primeiro segmento contém os itens atualmente na solução, enquanto o segundo segmento contém os itens que não estão na solução. O algoritmo gerencia dinamicamente a fronteira entre esses dois segmentos à medida que os itens são adicionados ou removidos da solução. Com essa estrutura de dados, a iteração pode ser feita rapidamente sobre o conjunto de itens que estão dentro ou fora da solução, além de verificar se um item está dentro da solução ou não. Quando um item é adicionado à solução, ele é movido para o início do segundo segmento. Em seguida, o índice que marca o final do primeiro segmento é aumentado, "deslocando" o primeiro elemento do segundo segmento para o final do primeiro segmento. A mesma ideia é seguida ao remover um item, movendo-o para a fronteira entre os dois segmentos e ajustando o índice que marca a separação. Essa atualização é feita em $O(1)$.

***Array* μ (mu)**: Este *array* contém valores que indicam como o valor total da solução mudará quando itens são adicionados ou removidos. Para um item i na solução, $\mu[i]$ armazena o valor pelo qual a solução é reduzida quando i é removido. Paralelamente, para um item j que não está na solução, $\mu[j]$ armazena o valor pelo qual o valor total da solução aumentará se j for adicionado. Em resumo, o *array* μ permite calcular o impacto da adição ou remoção de itens da solução em $O(1)$. Além disso, o *array* μ não está restrito por restrições de sinal, o que significa que adicionar ou remover um item pode aumentar, ou diminuir o valor da solução.

Inicialmente, para cada item i, $\mu[i] = p_i$. Quando i é removido da solução, os seguintes passos são realizados: para cada item conflitante j com i, o valor de $\mu[j]$ é incrementado em $d_{i,j}$. Ao adicionar um item i à solução, um processo semelhante é aplicado: itera-se sobre cada item j que forma um par com i, mas desta vez $\mu[j]$ é decrementado em $d_{i,j}$. Essa atualização tem complexidade $O(\Delta)$, onde Δ representa o maior número de conflitos que qualquer item único na instância tem com outros itens.

Lista Tabu TL: Este *array* indica quando um item se torna elegível para remoção da solução. Sua atualização é direta e tem complexidade $O(1)$: sempre que um item i é inserido na solução, $TL[i]$ é atualizado para indicar que i não pode ser removido pelas próximas TT iterações (definimos $TL[i] =$ iteração atual $+ TT$), onde TT é um parâmetro conhecido como *tabu tenure*. O seu valor é calibrado juntamente com outros parâmetros usados pelo método (Seção 5.3). A lista tabu dita o comportamento do algoritmo para evitar que os operadores se movam consistentemente na mesma direção, fazendo com que fique mais difícil ficar preso em um ótimo local (assim, diminuindo a dependência da perturbação). Outro benefício é acelerar a execução. Considerando que 3 dos 4 operadores de vizinhança envolvem a remoção de itens, reduzir o número de candidatos para remoção diminui o tempo necessário para encontrar uma melhoria, acelerando assim a busca local na totalidade.

***Heaps* H_S e H_X**: Dois *heaps* são utilizados na abordagem proposta: um *heap* é dedicado a armazenar itens já presentes na solução (H_S), enquanto o outro *heap* contém itens que ainda não foram incluídos na solução (H_X). Ambos foram estruturados como *max-heaps* sendo ordenados com base no peso dos itens. Essa configuração desempenha

um papel crucial durante a operação `swap-21`, pois nos permite implementar um mecanismo de filtragem que ajuda a evitar a avaliação de pares de itens que provavelmente não resultarão em soluções melhores. Esses *heaps* são atualizados usando métodos usuais de remoção/inserção em *heap*, que têm complexidade $O(\log(n))$.

4.2.2 Vizinhanças `swap-01` e `swap-10`

Para explorar a vizinhança `swap-01` (adição), o algoritmo itera sobre os itens fora da solução, verificando o valor de cada item i no *array* μ. Se $\mu[i] > 0$ e adicionar i não exceder a capacidade da mochila, o item i pode ser adicionado. Da mesma forma, para explorar a vizinhança `swap-10` (remoção), o algoritmo itera sobre os itens na solução. Um item i na solução pode ser removido se as seguintes condições forem atendidas: $TL[i] >$ iteração atual (o item i é elegível para remoção de acordo com a lista tabu TL) e $\mu[i] < 0$. Podemos ver que, usando o *array* μ e o *array* de solução, tanto `swap-01` quanto `swap-10` podem ser explorados em $O(n)$. Isso permite uma exploração eficiente dessas vizinhanças sem ter um impacto considerável por conta do tamanho do problema.

4.2.3 Vizinhança `swap-11`

Para o `swap-11`, o algoritmo explora todos os pares de itens (i, j), de modo que adicionar i à solução e remover j da solução não exceda a capacidade da mochila. Ao avaliar a remoção do item j, é necessário primeiro verificar se $TL[j] >$ `iteração_atual`. Itens que não atendem a essa condição não são considerados. Para cada par de itens considerado, o algoritmo verifica se a troca resulta em uma solução melhor. Isso é verificado quando a seguinte condição é satisfeita: $\mu[i] + d_{i,j} > \mu[j]$. A complexidade resultante dessa vizinhança é $O(n^2)$. Isso significa que a exploração completa de `swap-11` envolve comparar todos os pares possíveis de itens, tornando-a mais computacionalmente intensiva em comparação com as vizinhanças `swap-01` e `swap-10`.

4.2.4 Vizinhança `swap-21`

O Algoritmo 11 apresenta o pseudocódigo para explorar a vizinhança `swap-21`, que tenta melhorar a solução removendo dois itens $\{j, k\}$ da solução e adicionando um item i a ela. O algoritmo utiliza loops aninhados para examinar se uma troca com esse trio (i, j, k) pode ser realizada.

Algoritmo 11: `swap-21`. Recebe como entrada o conjunto de itens X, uma solução S e a iteração atual. Retorna uma solução atualizada S ou a mesma que a entrada.

```
1  Function swap-21(X, S, iteração_atual):
2      for j ∈ {j' : j' ∈ S, TL[j'] > iteração_atual, w_j' · f > H_X[0]} do
3          for k ∈ {k' : k' ∈ S, TL[k'] > iteração_atual} do
4              for i ∈ {i' : i' ∈ X \ S, w_i' ≤ w_j + w_k + capacidade_restante(S)} do
5                  entrada ← μ[i] + d_{j,i} + d_{k,i}
6                  saida ← μ[j] + μ[k] + d_{j,k}
7                  if entrada > saida then
8                      S ← S \ {j, k}
9                      S ← S ∪ {i}
10                     return S
11     return S
```

Para que `swap-21` execute com sucesso, os seguintes critérios devem ser satisfeitos: (i) j e k não devem estar na lista tabu, (ii) o valor $w_j \cdot f$ deve ser menor ou igual ao

peso do item mais pesado que não está na solução ($H_X[0]$). Aqui, f é um parâmetro do nosso algoritmo que serve como um filtro para excluir movimentos menos propensos a resultar em soluções melhoradas, e (iii) o peso do item i não deve exceder a capacidade restante da mochila quando consideramos a remoção de ambos j e k. Os itens a serem removidos, $\{j, k\}$, são avaliados em ordem com base em seu peso usando o *heap* H_S. É importante notar que, embora a complexidade no pior caso dessa vizinhança seja $O(n^3)$, a inclusão dos mecanismos de filtragem dentro dos loops aninhados (linhas 2, 3 e 4) reduz significativamente o tempo de execução final.

A linha 4 mostra a escolha dos itens candidatos à solução. Itens estes que não estão na solução e que quando considerado dentro da solução, não ultrapasse a capacidade da mochila (já considerando que os itens j e k já estão fora da mochila). As linhas 5 e 6 calculam em como o valor da solução será modificado. O valor de entrada, assim como no operador anterior, é o valor de adição do item i ignorando a existência das penalidades com os itens j e k (já que são candidatos à remoção, e que essas penalidades já foram subtraídas de $\mu[i]$). O valor de saída é calculado analogamente, como sendo o valor que a solução perde com a remoção dos itens e menos a penalidade entre eles (quando existe).

5 Resultados

Nesta seção, serão apresentados os resultados experimentais obtidos pelo algoritmo ILS-VND. Inicialmente, o ambiente no qual os testes foram conduzidos será estabelecido (Seção 5.1). Na sequência, as instâncias usadas (Seção 5.2)e os parâmetros usados no método (Seção 5.3). Em seguida, uma comparação direta com dois métodos já aplicados ao PMP (Seção 5.4), e, por fim, será analisado o efeito que as estruturas de dados propostas tiveram tanto na qualidade quanto no tempo de execução do método (Seção 5.5).

5.1 Ambiente e execução

O ILS-VND foi implementado em C++ e compilado usando o g++ 12.2.1 com *flags* de otimização "-O3". Os experimentos foram conduzidos em uma máquina com um processador Intel Celeron N4000 dual-core, em uma frequência base de 1,1 GHz, 4 MB de cache e 4 GB de RAM. O algoritmo foi executado em uma única *thread*, com apenas uma instância em execução por vez. O código-fonte do ILS-VND, bem como as instâncias consideradas, estão publicamente disponíveis.

As execuções do ILS-VND (exceto as execuções usadas para a análise da estrutura de dados em Seção 5.5) foram limitadas a um tempo de execução máximo de 10 segundos, com cada instância sujeita a 50 repetições, completamente aleatórias. A decisão de fixar o tempo máximo de execução do ILS-VND em 10 segundos foi influenciada pelo tempo mínimo relatado para o MA (D'AMBROSIO et al., 2023), especificamente 14,38 segundos (tempo médio para o grupo O-500).

5.2 Instâncias

As instâncias foram fornecidas pelos autores do algoritmo GA-CG (CAPOBIANCO et al., 2021). Essas instâncias estão divididas em três conjuntos, cada um com 40 instâncias: O, LK e MF.

- O conjunto O ("Original") consiste em 10 problemas de 500, 700, 800 e 1000 itens, totalizando 40 problemas. Para um problema com n itens, o número de pares penalizados l é igual a $6n$. Cada par envolve dois itens escolhidos aleatoriamente, e seu valor é um número inteiro escolhido aleatoriamente no intervalo [2, 15]. A capacidade da mochila b é igual a $3n$. O peso e o lucro de cada item também são escolhidos aleatoriamente nos intervalos [3, 20] e [5, 25], respectivamente.
- As instâncias do conjunto LK ("Large Knapsack") têm a mesma estrutura das instâncias do conjunto O, mas com uma capacidade aumentada de $b = 5n$.
- As instâncias do conjunto MF ("More Forfeits"') também têm a mesma estrutura das instâncias do conjunto O, mas desta vez incluem $l = 8n$ pares penalizados com os custos escolhidos da mesma forma que O.

5.3 Ajuste de parâmetros

Os parâmetros usados no ILS-VND foram determinados por meio de otimização bayesiana (SNOEK; LAROCHELLE; ADAMS, 2012). A Tabela 1 mostra o resultado do experimento da busca de parâmetros, onde a primeira coluna indica o parâmetro em questão, a segunda coluna indica o melhor valor e a terceira delimita o espaço de busca considerado para o parâmetro. O experimento foi realizado em um subconjunto das 15 instâncias mais difíceis até o então momento (todas as instâncias dos grupos O-1000,

Parametro	Encontrando	Intervalo de busca
c_1	1	[1, 25]
c_2	13	[1, 25]
c_3	1	[1, 25]
c_4	22	[1, 25]
f	1	[1, 25]
TT	1	[1, 700]

Tabela 1: Parâmetros encontrados para o ILS-VND.

O-800-02, MF-700-05, MF-800-07, MF-800-09, MF-1000-08), onde foi explorado dentre todos os valores inteiros dentro do respectivo espaço de busca de parâmetros.

O processo de otimização bayesiana aplicado considera como função objetivo a avaliação da soma dos valores de soluções para um determinado conjunto de instâncias, e como entrada para essa avaliação são considerados os parâmetros que se desejam otimizar. Esse processo de otimiza é iniciado com a seleção de alguns pontos iniciais e a avaliação da função objetivo nesses pontos. Com esses dados iniciais, um modelo probabilístico é construído, que serve para fazer previsões sobre o comportamento da função em pontos não observados. Este modelo então é atualizado iterativamente à medida que novos pontos são avaliados, refinando continuamente as estimativas da função objetivo, visto que a seleção desses novos pontos é guiada pela probabilidade de trazerem melhores resultados. O processo é encerrado quando todas as possibilidades são avaliadas ou ao ter um resultado satisfatório.

5.4 Resultados e Análise

Os resultados do ILS-VND foram comparados com os obtidos a partir do GA-CG (Seção 3.4.1) e MA (Seção 3.4.2). Como valor de referência, os resultados das instâncias apresentados no mesmo trabalho que o GA-CG (CAPOBIANCO et al., 2021) serão usados, o qual foi implementando usando modelo ILP (2.4.3).

O código-fonte do CAGG e do MA não pôde ser obtido, e, por isso, a comparação se baseia nos resultados relatados nas publicações de seus autores (CAPOBIANCO et al., 2021; D'AMBROSIO et al., 2023). Em seus respectivos trabalhos, foi indicado que as execuções do CAGG e do MA foram ambas limitadas a 150 iterações. Os resultados para o CPLEX também foram obtidos a partir do artigo do CAGG, como já dito anteriormente, onde cada execução do CPLEX foi limitada a 3 horas. Como referência, os resultados para CPLEX, GA-CG e MA usaram o mesmo ambiente: uma CPU Intel Xeon E5-2650 v3 de 2,3 GHz com 128 GB de RAM (CAPOBIANCO et al., 2021; D'AMBROSIO et al., 2023). Esses algoritmos foram implementados em C++, e é pertinente dizer que eles foram executados em uma máquina significativamente mais poderosa do que a que o ILS-VND foi executada.

Nas Tabelas 2, 3 e 4 apresentamos uma comparação numérica, instancia por instância ("ID"), entre CPLEX, GA-CG e ILS-VND, para cada conjunto de instâncias, agrupados por números de itens (500, 700, 800 e 1000). Nessas tabelas, a coluna "BEST" indica a melhor solução alcançada pelo respectivo método, "AVG." representa o valor médio das 50 soluções encontradas pelo ILS-VND, e "t(s)" indica o tempo de execução do CAGG. É importante lembrar que o tempo de execução do ILS-VND foi limitado a 10 segundos. Um asterisco é colocado sobre a solução quando esta é comprovadamente ótima.

Também usamos o gap como um parâmetro comparativo, uma vez que cobre o papel de ser um valor relativo à solução de referência, que, para este trabalho, é o valor obtido pelo CPLEX. O seu valor é dado pela seguinte expressão:

$$\text{gap} = 100\frac{sol_{ref} - sol_{fnd}}{sol_{ref}}$$

onde sol_{ref} é o valor de referência e sol_{fnd} é o resultado encontrado para o qual se deseja calcular o gap. Ainda sobre as Tabelas 2, 3 e 4, as colunas "GAP" referem-se aos gap calculados usando os melhores valores encontrados, e "'GAP_{avg}" refere-se ao gap calculado usando o valor médio das 50 soluções encontradas pelo ILS-VND. Como um último detalhe, também foi colocada uma linha "Mean" ao fim de cada tabela, trazendo a média de cada coluna referente a um gap (gap do GA-CG, gap do melhor valor do ILS-VND e gap do valor médio do ILS-VND). O MA foi omitido para estas tabelas por apresentar apenas resumos por grupo e tamanho de instâncias, os quais serão apresentados mais a frente.

Comparando o GA-CG ao ILS-VND, pode ser visto que o ILS-VND se sai melhor que o GA-CG em todas as instâncias. Se considerarmos que o valor médio da solução produzido pelo ILS-VND em suas 50 execuções (coluna "AVG") é o que pode ser esperado em uma única execução de 10 segundos, a diferença na eficiência se torna ainda mais evidente, já que o valor médio obtido pelo ILS-VND é superior ao resultado do GA-CG em todas as instâncias. Em comparação com o CPLEX, as melhores soluções obtidas pelo ILS-VND são iguais ou melhores do que as encontradas pelo CPLEX em 79 das 120 instâncias (coluna "BEST"). Essa conquista se torna um pouco mais interessante quando consideramos o tempo de execução: 3 horas para o CPLEX em comparação com um total de 8,3 minutos para o ILS-VND (10 segundos para cada uma das 50 execuções).

Como mostrado na Tabela 2, o CPLEX pôde resolver de forma ótima 47,5% das instâncias O, especialmente as menores com 500 e 700 itens. Para o GA-CG, o tempo de computação GA-CG aumentou mais de nove vezes quando consideramos 500 e 1000 itens, enquanto o gap permaneceu relativamente consistente, com uma média de 1,664% entre todo o conjunto. Em contraste, a média do gap do valor médio (média da coluna "GAP_{avg}") para o ILS-VND foi de 0,244%, mas aumentou à medida que o número de itens aumentou, com -0,04% sendo o melhor resultado e 0,545% o pior.

Considerando os resultados do conjunto de dados LK (Tabela 3) nota-se que nenhuma das soluções foi provada como ótima, e que o GA-CG obteve um gap médio de 1,7%, o que é comparável ao seu desempenho no conjunto O. No entanto, o método exigiu ainda mais tempo de execução para obter esses resultados. Em contraste, considerando a coluna "GAP", o ILS-VND encontrou soluções melhores que o CPLEX em 34 instâncias deste conjunto, resultando em uma média da coluna de -0,346%.

Por fim, os resultados do conjunto MF (Tabela 4) mostram que nenhuma das soluções foi certificada como ótima. O GA-CG manteve um gap médio de 1,767%, similar aos outros conjuntos, e com tempos de execução também semelhantes. Já o ILS-VND supera o CPLEX em 36 das 40 instâncias, considerando os resultados da coluna "BEST", apresentando como média das colunas valores negativos, o que significa que o ILS-VND foi capaz de obter melhores soluções tanto quando escolhemos a melhor solução dentro dos 8,3 minutos de execução total ou quando consideramos apenas o valor esperado para uma única execução de 10 segundos.

Incluindo agora os resultados do método MA, a Tabela 5 apresenta um sumário para cada grupo de instâncias. Nesta tabela, a coluna "GRUPO" indica qual conjunto de instâncias e número de itens está sendo tratado. As demais tabelas trazem o gap para seu respectivo método, com o tempo de execução médio entre parenteses. O tempo de execução para o ILS-VND foi omitido por ser fixado em 10 segundos.

Observa-se a superioridade do ILS-VND sobre o MA (e consequentemente sobre o GA-CG) quando é considerado que o ILS-VND atingiu um melhor valor médio em todos os

		CPLEX	**GA-CG**			**ILS-VND**			
	ID	BEST	BEST	t(s)	GAP	BEST	AVG.	GAP	GAP_{avg}
O-500	01	2626*	2568	165	2,209	2625	2622	0,152	0,152
	02	2660*	2621	159	1,466	2660	2657	0,000	0,113
	03	2516*	2478	162	1,510	2515	2507	0,040	0,358
	04	2556*	2515	167	1,604	2556	2552	0,000	0,156
	05	2625*	2582	166	1,638	2625	2622	0,000	0,114
	06	2615*	2557	155	2,218	2615	2612	0,000	0,115
	07	2627*	2602	163	0,952	2627	2625	0,000	0,076
	08	2556*	2522	162	1,330	2556	2552	0,000	0,156
	09	2613*	2572	169	1,569	2613	2611	0,000	0,077
	10	2558*	2537	165	0,821	2557	2554	0,039	0,156
O-700	01	3589*	3511	517	2,173	3588	3580	0,028	0,251
	02	3422	3359	510	1,841	3423	3415	**-0,029**	0,205
	03	3679*	3634	507	1,223	3678	3673	0,027	0,163
	04	3664*	3605	495	1,610	3663	3655	0,027	0,246
	05	3647*	3619	494	0,768	3647	3639	0,000	0,219
	06	3596*	3553	498	1,196	3595	3591	0,028	0,139
	07	3542*	3446	513	2,710	3541	3535	0,028	0,198
	08	3619*	3545	501	2,045	3615	3608	0,111	0,304
	09	3553	3487	508	1,858	3551	3540	0,056	0,366
	10	3652*	3594	515	1,588	3648	3640	0,110	0,329
O-800	01	4184	4125	789	1,410	4180	4170	0,096	0,335
	02	4065	4006	793	1,451	4064	4051	0,025	0,344
	03	4102	4018	796	2,048	4102	4094	0,000	0,195
	04	4051	3960	828	2,246	4050	4040	0,025	0,272
	05	4085	4041	803	1,077	4085	4077	0,000	0,196
	06	4249	4184	792	1,530	4247	4237	0,047	0,282
	07	4121	4021	803	2,427	4124	4110	**-0,073**	0,267
	08	4063*	4019	782	1,083	4060	4052	0,074	0,271
	09	4080	4017	792	1,544	4073	4062	0,172	0,441
	10	4124	4074	800	1,212	4129	4121	**-0,121**	0,073
O-1000	01	4927	4834	1597	1,888	4935	4920	**-0,162**	0,142
	02	4966	4893	1564	1,470	4985	4968	**-0,383**	**-0,040**
	03	5171	5070	1645	1,953	5172	5151	**-0,019**	0,387
	04	5141	5065	1553	1,478	5135	5120	0,117	0,408
	05	5134	5049	1526	1,656	5123	5106	0,214	0,545
	06	5082	4951	1589	2,578	5074	5059	0,157	0,453
	07	5100	5033	1585	1,314	5111	5089	**-0,216**	0,216
	08	5178	5071	1572	2,066	5180	5166	**-0,039**	0,232
	09	5108	5011	1640	1,899	5109	5086	**-0,020**	0,431
	10	5178	5080	1590	1,893	5169	5155	0,174	0,444
Mean					1,664			0,014	0,244

Tabela 2: Tabela com a execução das instâncias O

		CPLEX	**GA-CG**			**ILS-VND**			
	ID	BEST	BEST	t(s)	GAP	BEST	AVG.	GAP	GAP_{avg}
LK-500	01	2712	2649	238	2,323	2719	2716	**-0,258**	**-0,147**
	02	2729	2666	222	2,309	2739	2737	**-0,366**	**-0,293**
	03	2639	2587	209	1,970	2639	2637	0,000	0,076
	04	2665	2600	222	2,439	2665	2663	0,000	0,075
	05	2686	2615	218	2,643	2689	2685	**-0,112**	0,037
	06	2746	2707	224	1,420	2754	2750	**-0,291**	**-0,146**
	07	2689	2659	226	1,116	2706	2700	**-0,632**	**-0,409**
	08	2681	2644	219	1,380	2681	2680	0,000	0,037
	09	2652	2615	223	1,395	2652	2648	0,000	0,151
	10	2665	2619	227	1,726	2675	2672	**-0,375**	**-0,263**
LK-700	01	3757	3678	682	2,103	3760	3752	**-0,080**	0,133
	02	3611	3535	695	2,105	3612	3604	**-0,028**	0,194
	03	3824	3799	691	0,654	3833	3822	**-0,235**	0,052
	04	3835	3749	688	2,243	3840	3833	**-0,130**	0,052
	05	3823	3750	639	1,909	3842	3838	**-0,497**	**-0,392**
	06	3707	3651	673	1,511	3719	3713	**-0,324**	**-0,162**
	07	3676	3624	704	1,415	3684	3676	**-0,218**	0,000
	08	3762	3735	667	0,718	3796	3789	**-0,904**	**-0,718**
	09	3651	3603	702	1,315	3650	3643	0,027	0,219
	10	3832	3736	697	2,505	3832	3830	0,000	0,052
LK-800	01	4298	4213	1101	1,978	4308	4298	**-0,233**	0,000
	02	4201	4129	1021	1,714	4217	4204	**-0,381**	**-0,071**
	03	4251	4128	1115	2,893	4255	4244	**-0,094**	0,165
	04	4209	4146	1116	1,497	4216	4199	**-0,166**	0,238
	05	4176	4145	1060	0,742	4228	4220	**-1,245**	**-1,054**
	06	4417	4343	1077	1,675	4420	4414	**-0,068**	0,068
	07	4284	4175	1118	2,544	4293	4282	**-0,210**	0,047
	08	4144	4120	1138	0,579	4158	4150	**-0,338**	**-0,145**
	09	4271	4225	1087	1,077	4301	4293	**-0,702**	**-0,515**
	10	4277	4192	1097	1,987	4279	4272	**-0,047**	0,117
LK-1000	01	5147	5037	2214	2,137	5174	5162	**-0,525**	**-0,291**
	02	5156	5025	2122	2,541	5160	5143	**-0,078**	0,252
	03	5340	5281	2125	1,105	5394	5372	**-1,011**	**-0,599**
	04	5421	5323	2087	1,808	5430	5414	**-0,166**	0,129
	05	5254	5209	2138	0,856	5306	5282	**-0,990**	**-0,533**
	06	5345	5234	2029	2,077	5353	5325	**-0,150**	0,374
	07	5244	5139	2159	2,002	5266	5248	**-0,420**	**-0,076**
	08	5323	5245	2005	1,465	5383	5354	**-1,127**	**-0,582**
	09	5295	5225	2042	1,322	5342	5323	**-0,888**	**-0,529**
	10	5362	5318	2060	0,821	5394	5377	**-0,597**	**-0,280**
Mean					1,700			-0,346	-0,118

Tabela 3: Tabela com a execução das instâncias LK

		CPLEX	**GA-CG**			**ILS-VND**			
	ID	BEST	BEST	t(s)	GAP	BEST	AVG.	GAP	GAP_{avg}
MF-500	01	2368	2305	253	2,660	2367	2364	0,042	0,169
	02	2310	2300	256	0,433	2318	2312	**-0,346**	**-0,087**
	03	2284	2229	254	2,408	2284	2280	0,000	0,175
	04	2259	2228	247	1,372	2270	2263	**-0,487**	**-0,177**
	05	2321	2272	263	2,111	2321	2316	0,000	0,215
	06	2316	2283	254	1,425	2322	2317	**-0,259**	**-0,043**
	07	2288	2207	250	3,540	2290	2285	**-0,087**	0,131
	08	2201	2161	248	1,817	2215	2210	**-0,636**	**-0,409**
	09	2259	2219	256	1,771	2263	2261	**-0,177**	**-0,089**
	10	2305	2285	254	0,868	2310	2307	**-0,217**	**-0,087**
MF-700	01	3127	3050	752	2,462	3127	3119	0,000	0,256
	02	3038	2966	735	2,370	3053	3043	**-0,494**	**-0,165**
	03	3197	3162	758	1,095	3225	3214	**-0,876**	**-0,532**
	04	3233	3176	753	1,763	3247	3239	**-0,433**	**-0,186**
	05	3238	3134	723	3,212	3246	3235	**-0,247**	0,093
	06	3129	3095	733	1,087	3133	3129	**-0,128**	0,000
	07	3015	2948	745	2,222	3048	3036	**-1,095**	**-0,697**
	08	3166	3096	724	2,211	3174	3168	**-0,253**	**-0,063**
	09	3186	3146	753	1,255	3214	3194	**-0,879**	**-0,251**
	10	3203	3154	726	1,530	3216	3208	**-0,406**	**-0,156**
MF-800	01	3691	3639	1132	1,409	3702	3692	**-0,298**	**-0,027**
	02	3711	3647	1114	1,725	3723	3717	**-0,323**	**-0,162**
	03	3605	3566	1195	1,082	3668	3657	**-1,748**	**-1,442**
	04	3490	3391	1097	2,837	3527	3505	**-1,060**	**-0,430**
	05	3741	3704	1097	0,989	3752	3743	**-0,294**	**-0,053**
	06	3772	3697	1106	1,988	3775	3769	**-0,080**	0,080
	07	3683	3611	1100	1,955	3689	3675	**-0,163**	0,217
	08	3575	3524	1118	1,427	3612	3606	**-1,035**	**-0,867**
	09	3593	3521	1082	2,004	3600	3588	**-0,195**	0,139
	10	3633	3531	1114	2,808	3649	3631	**-0,440**	0,055
MF-1000	01	4450	4393	2200	1,281	4463	4450	**-0,292**	0,000
	02	4408	4337	2053	1,611	4467	4451	**-1,338**	**-0,975**
	03	4577	4572	2102	0,109	4630	4610	**-1,158**	**-0,721**
	04	4564	4468	2107	2,103	4584	4564	**-0,438**	0,000
	05	4468	4399	2106	1,544	4519	4502	**-1,141**	**-0,761**
	06	4569	4512	2073	1,248	4597	4580	**-0,613**	**-0,241**
	07	4578	4485	2162	2,031	4608	4594	**-0,655**	**-0,349**
	08	4486	4421	2123	1,449	4570	4543	**-1,872**	**-1,271**
	09	4631	4570	2036	1,317	4640	4630	**-0,194**	0,022
	10	4660	4559	2096	2,167	4664	4651	**-0,086**	0,193
Mean					1,767			-0,510	-0,212

Tabela 4: Tabela com a execução das instâncias MF

GRUPO	GA-CG	MA	ILS-VND
O-500	1,53 (163,90s)	0,62 (14,38s)	0,15
O-700	1,70 (506,22s)	0,90 (56,05s)	0,24
O-800	1,60 (798,20s)	0,91 (111,30s)	0,27
O-1000	1,82 (1586,59s)	0,80 (312,24s)	0,32
LK-500	1,87 (233,44s)	0,37 (33,64s)	**-0,09**
LK-700	1,65 (684,33s)	0,43 (139,39s)	**-0,06**
LK-800	1,67 (1093,46s)	0,27 (318,64s)	**-0,11**
LK-1000	1,61 (2098,57s)	0,18 (719,10s)	**-0,21**
MF-500	1,84 (253,96s)	0,51 (40,20s)	**-0,02**
MF-700	1,92 (740,86s)	0,34 (171,98s)	**-0,17**
MF-800	1,82 (1115,93s)	0,29 (236,00s)	**-0,25**
MF-1000	1,49 (2106,27s)	-0,19 (551,51s)	**-0,41**

Tabela 5: Tabela com o gap do valor médio e o tempo de execução para cada grupo nos conjuntos. Resultados em negrito indicam uma solução melhor do que o CPLEX. O tempo de execução do ILS-VND foi omitido já que é fixado em 10 segundos.

grupos apresentados na Tabela 5, em um tempo de execução significativamente menor. O grupo no qual o MA mais se aproxima do ILS-VND é no grupo LK-800, com uma diferença no gap de 0,38%, mas quando reparamos que o MA levou 308 segundos a mais para obter esse resultado (que já é inferior), essa diferença reduzida, quando comparada as demais diferenças entre gaps, se torna menos interessante.

É importante ressaltar que nossos experimentos (todas as execuções do ILS-VND) foram conduzidos em um computador significativamente menos capaz que o que foi usado nos experimentos do MA, GA-CG e CPLEX. Novamente, e, apesar dessa diferença, o ILS-VND apresentou resultados superiores aos outros métodos em todos os conjuntos. Além de obter tais resultados, o ILS-VND também pode ser considerado uma melhor escolha quando recursos computacionais são mais restritos.

5.5 Impacto dos componentes e estruturas de dados

Também analisamos o efeito de alguns dos componentes e estruturas de dados do ILS-VND apresentados na Seção 4.2. Inicialmente, avaliamos a eficácia das estruturas de dados de heap, onde executamos o ILS-VND com e sem a funcionalidade de filtragem de vértices que recorrem aos heaps em todas as instâncias, considerando diferentes limites de tempo: 1s, 10s e 100s.

A Figura 2 revela o efeito dos *heaps* no número de iterações do ILS-VND em função do número de itens. Exceto por algumas instâncias, existe uma distinção clara no número de iterações alcançadas com e sem o uso de *heaps*. Também podemos observar que o número de iterações tende a convergir à medida que o número de itens aumenta, o que pode indicar uma limitação desse aprimoramento para um certo número de itens. Em outras palavras, além de um determinado limiar, a vantagem proporcionada pelo uso de *heap* pode se tornar insignificante.

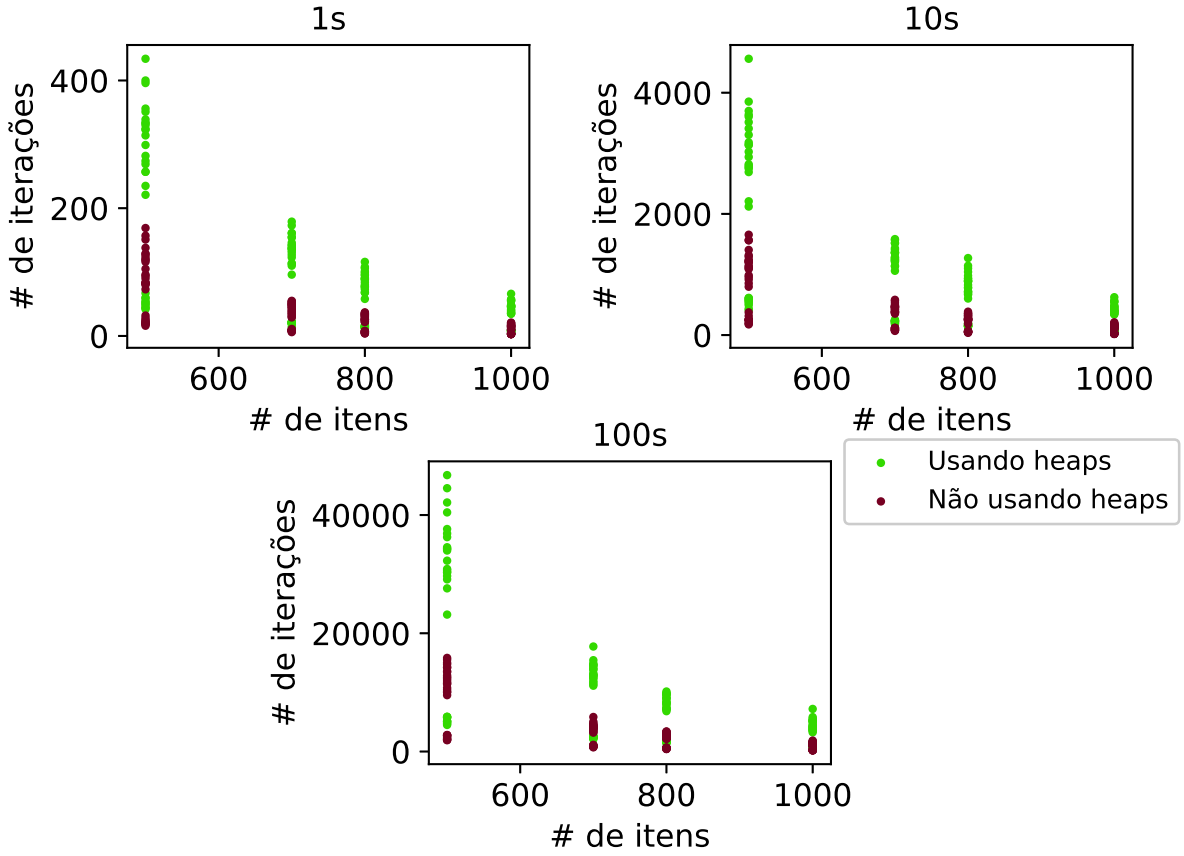


Figura 2: Efeito do uso das heaps no número de iterações do ILS-VND.

O segundo experimento apresenta uma avaliação detalhada da melhora induzida por cada estrutura de dados. Neste experimento, consideramos cinco versões distintas do algoritmo. A Tabela 6 mostra os componentes de cada uma dessas versões, e a Tabela 7 apresenta uma comparação entre elas considerando a média do gap do valor médio. Cada versão traz progressivamente um componente, o quais foram descritos na Seção 4.2.

A análise se concentrou nos grupos de instâncias mais difíceis (grupos onde o ILS-VND obteve os piores resultados em sua primeira versão), os mesmos utilizados para ajuste, como explicado na Subseção 5.3. Cada iteração apresentada na Tabela 7 representa uma etapa de desenvolvimento no processo do ILS-VND.

Versão	Heaps	`swap-11`	`swap-21`	`swap-01`	`swap-10`	Tabu List
Base		✓	✓			
Heap	✓	✓	✓			
Novos swaps	✓			✓	✓	
Todos swaps	✓	✓	✓	✓	✓	
Tabu	✓	✓	✓	✓	✓	✓

Tabela 6: Componentes usados em diferentes versões do ILS-VND.

A segunda versão (Coluna 3 na Tabela 7), que inclui a filtragem com auxílio de heaps, foi a primeira versão que produziu resultados razoavelmente satisfatórios (superando o GA-CG). Esta versão utilizou exclusivamente as estruturas de vizinhança `swap-11` e `swap-21`, juntamente com os heaps. Na terceira versão, as estruturas de vizinhança `swap-01` e `swap-10` foram introduzidas inicialmente como substitutas para o `swap-11` e

`swap-21`, o que acabou resultando em um pior desempenho. A quarta versão conseguiu superar os resultados da segunda versão reintroduzindo as estruturas de vizinhança `swap-11` e `swap-21`, mas agora dividindo a busca local em duas fases, uma para cada grupo de estruturas de vizinhança (como apresentado na Seção 4.1.2). Por fim, a quinta versão, que incorporou a Lista Tabu, marcou o ponto de obtenção dos melhores resultados até então. Pode ser observada uma melhoria na qualidade da solução por meio da adição/melhoria das estruturas de dados e das estruturas de vizinhança. Na verdade, existe uma redução de aproximadamente 93% no gap no cenário de melhor caso (O-800) e aproximadamente 80% no cenário de pior caso (O-1000), desde a versão inicial até a última.

Grupo	Base	Heap	Novos swaps	Todos swaps	Tabu
O-800	4,28	0,44	2,01	0,71	0,27
O-1000	2,57	0,44	1,77	0,60	0,32
MF-700	2,13	0,77	0,30	0,55	-0,17
MF-800	3,06	0,77	0,08	0,05	-0,25
MF-1000	2,13	-0,44	-0,84	-0,22	-0,41

Tabela 7: Redução do gap do valor médio obtido em 5 versões diferentes do ILS-VND.

6 Considerações Finais

A classe de problemas pertencentes à família dos problemas da mochila é extremamente estudada por sua capacidade de representar diversos outros problemas em diversas áreas; de problemas financeiros, logísticos e gerência de projetos. Dentre as diversas variações do problema, pode ser citada sua forma clássica, o Problema da Mochila 0-1, ou ainda, com a adição de pares proibidos, o Problema da Mochila Disjunta, e, ao invés de proibir, penalizar os pares, temos o Problema da Mochila com Penalidades. Em resumo, este problema consiste em selecionar um conjunto de itens, onde cada item tem um valor e um peso. O objetivo é selecionar um subconjunto de itens que maximize a soma dos valores sem ultrapassar a capacidade de peso da mochila. Entretanto, certo pares de itens reduzem o valor total da mochila, sendo estes os pares penalizados.

Os métodos exatos para grandes instâncias se tornam inviáveis pela complexidade do problema, NP-difícil, portanto, justifica-se o uso de heurísticas e meta-heurísticas, das quais se pode citar o Carrossel Guloso. GA-CG e MA, todos já aplicados ao Problema da Mochila com Penalidades; ou ainda um método hibrido ILS-VND, o qual já foi aplicado com sucesso em problemas com grafos relacionados com o problema da mochila.

Este trabalho introduziu o ILS-VND para o Problema da Mochila com Penalidade. A busca local do ILS-VND considera quatro estruturas de vizinhança: `swap-01`, `swap-10`, `swap-11` e `swap-21`. Essas vizinhanças são exploradas de forma sistemática usando um procedimento VND juntamente com uma abordagem *first improvement*. Para uma exploração rápida dessas vizinhanças, o método proposto adota estruturas de dados especializadas, como estruturas de *heap*, e *array* de valores pré-calculados. Além disso, uma lista tabu é utilizada para melhorar a qualidade da solução e evitar que a busca fique presa em ótimos locais.

O ILS-VND foi avaliado em comparação com três métodos da literatura: MA, GA-CG e um modelo ILP. Os experimentos computacionais envolvendo três conjuntos de instâncias obtidos da literatura apontaram que o novo método supera outras heurísticas tanto em termos de qualidade da solução quanto de tempo computacional. Os resultados mostraram a superioridade do ILS-VND em relação aos demais métodos não exatos em quesito de qualidade da solução. Isso é evidenciado uma vez que o ILS-VND trouxe os melhores resultados em todas as instâncias, e, quando não atingindo os ótimos conhecidos, chegando muito próximo ou, na maioria dos casos para dois dos três conjuntos de instâncias, superando o método exato. Considerando as médias dos valores obtidos, o ILS-VND é uma ótima escolha quando temos um problema denso (muitos pares penalizados) ou uma mochila com muita capacidade, fazendo com que o número de itens elegíveis para a solução se mantenha alto, tornando o problema mais computacionalmente custoso.

Observando da qualidade potencial das soluções entregues pelo ILS-VND, o método pode ser tornar ainda mais atrativo quando o seu tempo de execução é considerado para os resultados obtidos, sendo 10 segundos para uma única instância, ou aproximadamente 8,3 minutos para a executar uma única instância 50 vezes e extraindo a melhor solução. O desempenho é notório quando se é lembrado que, para o método exato, foi dado 3 horas de execução por instância, sendo considerado o resultado final a melhor solução encontrada, o qual foi comparado com a nossa melhor solução encontrada em 8,3 minutos. Já para os outros métodos considerados nenhum tempo menor ou igual a 10 segundos foi reportado, o que somado com a melhor qualidade de soluções do ILS-VND, reafirma a posição de superioridade em relação aos demais.

Também foram realizados experimentos computacionais adicionais para elucidar o impacto de componentes específicos e estruturas de dados dentro do método proposto. Um primeiro experimento considerou o uso de heaps reduzir o tempo de execução de uma única

iteração do método, aumentando assim o número de iterações em um mesmo período de tempo. As heaps são usadas em um filtro que reduz o número de itens considerados para a avaliação do `swap-21`, e como um dos resultados obtidos, o número de iterações foi aproximadamente dobrado para instâncias com poucos itens. Outro experimento considerou as diferentes versões do método durante sua evolução, onde algumas combinações de componentes foram consideradas, como diferentes combinações dos quatro operadores de vizinhanças usados. Usando como referência a primeira versão do método, em relação à última versão, o ganho de qualidade da solução ficou entre 80% e 93% nas instâncias testadas, o que é um ganho significativo.

Com isso, podemos observar que a implementação ILS-VND apresentada aqui é altamente competitiva com os métodos já existentes para o Problema da Mochila com Penalidades (PMP). No entanto, é necessário fazer mais, como tentar reduzir sua complexidade espacial. Por exemplo, há redundâncias na forma como os itens são armazenados. Também é importante considerar uma forma de desvincular o método das estruturas de dados que o auxiliam, o que abre mais possibilidades para o algoritmo lidar com a seleção de itens dentro das vizinhanças ou realizar análises mais complexas no critério de seleção.

A resolução desse problema de forma mais eficiente traz diversos benefícios. Um exemplo simples e sem necessidade de adaptações é a capacidade de trabalhar com instâncias ainda maiores do que se é considerado o usual na literatura, ou seja, instancias com número de itens que se executadas nos métodos tidos até então, demoraria ordens de grandeza a mais para encontrar uma solução de qualidade similar ao ILS-VND. Outro exemplo de impacto direto é a economia de energia e recursos em geral. Imaginando uma situação hipotética em que um grande volume de dados deve ser processados, e que durante esse processamento surgem subproblemas que enquadram no PMP, um método capaz de trazer soluções satisfatórias em um tempo menor, reduz bastante o custo computacional total para o processamento. Temos também o benefício de existir mais um método para outros problemas relacionados. Como já abordado, diversos problemas são semelhantes ao PMP, e com pouco esforço se pode transformar uma instância de algum desses outros problemas em uma instância compatível para o nosso método.

Por outro lado, existem pontos que tornam o ILS-VND vulnerável a métodos mais eficientes. O primeiro ponto de vulnerabilidade também é um benefício já discutido, a transformação de uma mesma instancia entre problemas similares. Considerando outro problema similar, podemos transformar uma instância do PMP para esse outro problema, aplicar um método, e transformar a solução obtida para uma solução do PMP. Caso o custo de ambas transformações mais o tempo de execução do método for menor que o do ILS-VND, o nosso método deixa de ser tão competitivo.

Ainda olhando para esta transitividade entre problemas, futuramente espera-se aplicar o método em instâncias de outros problemas, ampliando o escopo e analisando o desempenho em comparação com outros métodos. Exemplos de problemas que podem ser testados é o MWIS (NOGUEIRA; PINHEIRO; SUBRAMANIAN, 2017), a versão disjunta do problema (PFERSCHY; SCHAUER, 2009) ou *scheduling* (BAKER; JR, 1996).

Pesquisas futuras incluem a análise de vizinhanças de busca local adicionais e mais estruturas de dados alternativas para uma exploração ainda mais rápida. Além disso, o desempenho do algoritmo pode ser potencialmente aprimorado incorporando componentes de algoritmos projetados para problemas semelhantes, como o caso da oscilação estratégica (WEI et al., 2023), onde se pode considerar soluções viáveis que sejam "vizinhas" a soluções inviáveis. Também é possível expandir o método para o problema onde podem existir conjuntos penalizados com mais de dois itens (D'AMBROSIO et al., 2023), e para isso seria necessário, por exemplo, estudar formas eficientes de avaliação de trocas de itens, ou estruturas que quando não substituindo, auxiliem na busca de soluções melhores.

7 Referências

ABDEL-BASSET, M.; EL-SHAHAT, D.; SANGAIAH, A. K. A modified nature inspired meta-heuristic whale optimization algorithm for solving 0–1 knapsack problem. *International Journal of Machine Learning and Cybernetics*, Springer, v. 10, p. 495–514, 2019.

AKINC, U. Approximate and exact algorithms for the fixed-charge knapsack problem. *European Journal of Operational Research*, v. 170, n. 2, p. 363–375, 2006. ISSN 0377-2217. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0377221704005016>.

ARCHETTI, C.; BERTAZZI, L.; SPERANZA, M. G. Reoptimizing the 0–1 knapsack problem. *Discrete applied mathematics*, Elsevier, v. 158, n. 17, p. 1879–1887, 2010.

ATAMTüRK, A.; NEMHAUSER, G. L.; SAVELSBERGH, M. W. Conflict graphs in solving integer programming problems. *European Journal of Operational Research*, Elsevier BV, v. 121, n. 1, p. 40–55, Feb 2000. ISSN 0377-2217. Disponível em: <http://dx.doi.org/10.1016/S0377-2217(99)00015-6>.

AVIS, D.; HERTZ, A.; MARCOTTE, O. *Graph theory and combinatorial optimization*. [S.l.]: Springer Science & Business Media, 2005. v. 8.

AXIOTIS, K.; TZAMOS, C. Capacitated dynamic programming: Faster knapsack and graph algorithms. *arXiv preprint arXiv:1802.06440*, 2018.

BAKER, B. S.; JR, E. G. C. Mutual exclusion scheduling. *Theoretical Computer Science*, Elsevier, v. 162, n. 2, p. 225–243, 1996.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, v. 15, n. 2, p. 56–69, 1993.

BELLMAN, R. Some applications of the theory of dynamic programming—a review. *Journal of the Operations Research Society of America*, Institute for Operations Research and the Management Sciences (INFORMS), v. 2, n. 3, p. 275–288, Aug 1954. ISSN 2326-3229. Disponível em: <http://dx.doi.org/10.1287/opre.2.3.275>.

BELLMAN, R. Dynamic programming. *Science*, American Association for the Advancement of Science, v. 153, n. 3731, p. 34–37, 1966.

BETTINELLI, A.; CACCHIANI, V.; MALAGUTI, E. A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, Institute for Operations Research and the Management Sciences (INFORMS), v. 29, n. 3, p. 457–473, Aug 2017. ISSN 1526-5528. Disponível em: <http://dx.doi.org/10.1287/ijoc.2016.0742>.

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 35, n. 3, p. 268–308, Sep 2003. ISSN 1557-7341. Disponível em: <http://dx.doi.org/10.1145/937503.937505>.

BODLAENDER, H. L.; JANSEN, K. On the complexity of scheduling incompatible jobs with unit-times. In: SPRINGER. *International Symposium on Mathematical Foundations of Computer Science*. [S.l.], 1993. p. 291–300.

BOMZE, I. M. et al. The maximum clique problem. *Handbook of Combinatorial Optimization*, Springer US, p. 1–74, 1999. Disponível em: <http://dx.doi.org/10.1007/978-1-4757-3023-4_1>.

BRADLEY, G. H. Transformation of integer programs to knapsack problems. *Discrete Mathematics*, Elsevier BV, v. 1, n. 1, p. 29–45, May 1971. ISSN 0012-365X. Disponível em: <http://dx.doi.org/10.1016/0012-365X(71)90005-7>.

CAPOBIANCO, G. et al. A hybrid metaheuristic for the knapsack problem with forfeits. *Soft Computing*, Springer Science and Business Media LLC, v. 26, n. 2, p. 749–762, Oct 2021. ISSN 1433-7479. Disponível em: <http://dx.doi.org/10.1007/s00500-021-06331-x>.

CARRABS, F. et al. Column generation embedding carousel greedy for the maximum network lifetime problem with interference constraints. In: SPRINGER. *Optimization and Decision Science: Methodologies and Applications: ODS, Sorrento, Italy, September 4-7, 2017 47*. [S.l.], 2017. p. 151–159.

CERRONE, C.; CERULLI, R.; GOLDEN, B. Carousel greedy: A generalized greedy algorithm with applications in optimization. *Computers & Operations Research*, Elsevier BV, v. 85, p. 97–112, Sep 2017. ISSN 0305-0548. Disponível em: <http://dx.doi.org/10.1016/j.cor.2017.03.016>.

CERULLI, R. et al. Maximum network lifetime problem with time slots and coverage constraints: heuristic approaches. *The Journal of Supercomputing*, Springer, v. 78, n. 1, p. 1330–1355, 2022.

CERULLI, R. et al. The knapsack problem with forfeits. *Combinatorial Optimization*, Springer International Publishing, p. 263–272, 2020. ISSN 1611-3349. Disponível em: <http://dx.doi.org/10.1007/978-3-030-53262-8_22>.

CESELLI, A.; RIGHINI, G. An optimization algorithm for a penalized knapsack problem. *Operations Research Letters*, v. 34, n. 4, p. 394–404, 2006. ISSN 0167-6377. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167637705000751>.

CESELLI, A.; RIGHINI, G. An optimization algorithm for a penalized knapsack problem. *Operations Research Letters*, Elsevier BV, v. 34, n. 4, p. 394–404, Jul 2006. ISSN 0167-6377. Disponível em: <http://dx.doi.org/10.1016/j.orl.2005.06.001>.

COLOMBI, M. et al. The directed profitable rural postman problem with incompatibility constraints. *European Journal of Operational Research*, Elsevier, v. 261, n. 2, p. 549–562, 2017.

COLOMBI, M.; MANSINI, R.; SAVELSBERGH, M. The generalized independent set problem: Polyhedral analysis and solution approaches. *European Journal of Operational Research*, Elsevier BV, v. 260, n. 1, p. 41–55, Jul 2017. ISSN 0377-2217. Disponível em: <http://dx.doi.org/10.1016/j.ejor.2016.11.050>.

CONIGLIO, S.; FURINI, F.; SEGUNDO, P. S. A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *European Journal of Operational Research*, Elsevier BV, v. 289, n. 2, p. 435–455, Mar 2021. ISSN 0377-2217. Disponível em: <http://dx.doi.org/10.1016/j.ejor.2020.07.023>.

COOK, S. The p versus np problem. *Clay Mathematics Institute*, v. 2, 2000.

COOK, S. A. An overview of computational complexity. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 26, n. 6, p. 400–408, Jun 1983. ISSN 1557-7317. Disponível em: <http://dx.doi.org/10.1145/358141.358144>.

CROCE, F. D.; PFERSCHY, U.; SCATAMACCHIA, R. New exact approaches and approximation results for the penalized knapsack problem. *Discrete Applied Mathematics*, Elsevier BV, v. 253, p. 122–135, Jan 2019. ISSN 0166-218X. Disponível em: <http://dx.doi.org/10.1016/j.dam.2017.11.023>.

DANTZIG, G. *Linear Programming and Extensions*. Princeton University Press, 1963. ISBN 9781400884179. Disponível em: <http://dx.doi.org/10.1515/9781400884179>.

DANTZIG, G. B. Discrete-variable extremum problems. *Operations research*, INFORMS, v. 5, n. 2, p. 266–288, 1957.

DARMANN, A. et al. Paths, trees and matchings under disjunctive constraints. *Discrete Applied Mathematics*, Elsevier, v. 159, n. 16, p. 1726–1735, 2011.

Della Croce, F.; PFERSCHY, U.; SCATAMACCHIA, R. New exact approaches and approximation results for the penalized knapsack problem. *Discrete Applied Mathematics*, v. 253, p. 122–135, 2019. ISSN 0166-218X. 14th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW 2016). Disponível em: <https://www.sciencedirect.com/science/article/pii/S0166218X17305449>.

DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. In: *Secure communications and asymmetric cryptosystems*. [S.l.]: Routledge, 2019. p. 143–180.

D'AMBROSIO, C. et al. The knapsack problem with forfeit sets. *Computers & Operations Research*, Elsevier, v. 151, p. 106093, 2023.

FENG, Y. et al. Solving 0–1 knapsack problem by a novel binary monarch butterfly optimization. *Neural computing and applications*, Springer, v. 28, p. 1619–1634, 2017.

GHERBOUDJ, A.; CHIKHI, S. Bpso algorithms for knapsack problem. In: SPRINGER. *International Conference on Computer Networks and Communications*. [S.l.], 2011. p. 217–227.

GURSKI, F.; REHS, C. The knapsack problem with conflict graphs and forcing graphs of bounded clique-width. *Operations Research Proceedings 2018*, Springer International Publishing, p. 259–265, 2019. ISSN 2197-9294. Disponível em: <http://dx.doi.org/10.1007/978-3-030-18500-8_33>.

HANSEN, P. et al. Variable neighborhood search. *Handbook of Metaheuristics*, Springer International Publishing, p. 57–97, Sep 2018. ISSN 2214-7934. Disponível em: <http://dx.doi.org/10.1007/978-3-319-91086-4_3>.

HARTMANIS, J. Computers and intractability: A guide to the theory of np-completeness (michael r. garey and david s. johnson). *SIAM Review*, Society for Industrial and Applied Mathematics (SIAM), v. 24, n. 1, p. 90–91, Jan 1982. ISSN 1095-7200. Disponível em: <http://dx.doi.org/10.1137/1024022>.

HARTMANIS, J.; STEARNS, R. E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, v. 117, p. 285–306, 1965.

HIFI, M.; MICHRAFY, M. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society*, Taylor & Francis, v. 57, n. 6, p. 718–726, 2006.

HIFI, M.; MICHRAFY, M. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. *Computers & operations research*, Elsevier, v. 34, n. 9, p. 2657–2673, 2007.

HIFI, M.; MOUNIR, M. O. A. Un algorithme augmenté pour le problème du knapsack disjonctif. *ROADEF 2009*, p. 324, 2009.

HIFI, M.; OTMANI, N. An algorithm for the disjunctively constrained knapsack problem. *International Journal of Operational Research*, Inderscience Publishers, v. 13, n. 1, p. 22, 2012. ISSN 1745-7653. Disponível em: <http://dx.doi.org/10.1504/IJOR.2012.044026>.

HOCHBAUM, D. S. 50th anniversary article: Selection, provisioning, shared fixed costs, maximum closure, and implications on algorithmic methods today. *Management Science*, INFORMS, v. 50, n. 6, p. 709–723, 2004.

HOCHBAUM, D. S.; PATHRIA, A. Forest harvesting and minimum cuts: a new approach to handling spatial constraints. *Forest Science*, Oxford University Press, v. 43, n. 4, p. 544–554, 1997.

HOPKINS, E. Postal bodies: Imagining communication infrastructures in nineteenth-century literature. *University of Exeter*, University of Exeter, p. 60–72, 2021.

ILOG, I. User's manual for cplex. *International Business Machines Corporation*, v. 46, n. 53, p. 157, 2009.

JANSEN, K.; ÖHRING, S. Approximation algorithms for time constrained scheduling. *Information and computation*, Elsevier, v. 132, n. 2, p. 85–108, 1997.

KANTOROVICH, L. V. Mathematical methods of organizing and planning production. *Management science*, INFORMS, v. 6, n. 4, p. 366–422, 1960. Disponível em: <http://dx.doi.org/10.1287/mnsc.6.4.366>.

KARP, R. M. Reducibility among combinatorial problems. *Complexity of Computer Computations*, Springer US, p. 85–103, 1972. Disponível em: <http://dx.doi.org/10.1007/978-1-4684-2001-2_9>.

KOKASH, N. An introduction to heuristic algorithms. *Department of Informatics and Telecommunications*, Citeseer, p. 1–8, 2005.

KOLESAR, P. J. A branch and bound algorithm for the knapsack problem. *Management science*, INFORMS, v. 13, n. 9, p. 723–735, 1967.

KONG, X. et al. A simplified binary harmony search algorithm for large scale 0–1 knapsack problems. *Expert Systems with Applications*, Elsevier, v. 42, n. 12, p. 5337–5355, 2015.

LAWLER, E. L.; WOOD, D. E. Branch-and-bound methods: A survey. *Operations Research*, Institute for Operations Research and the Management Sciences (INFORMS), v. 14, n. 4, p. 699–719, Aug 1966. ISSN 1526-5463. Disponível em: <http://dx.doi.org/10.1287/opre.14.4.699>.

LIN, F.-T.; KAO, C.-Y.; HSU, C.-C. Applying the genetic approach to simulated annealing in solving some np-hard problems. *IEEE Transactions on systems, man, and cybernetics*, IEEE, v. 23, n. 6, p. 1752–1767, 1993.

LIN, S. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, Wiley Online Library, v. 44, n. 10, p. 2245–2269, 1965.

LOURENçO, H. R.; MARTIN, O. C.; STüTZLE, T. Iterated local search: Framework and applications. *Handbook of Metaheuristics*, Springer International Publishing, p. 129–168, Sep 2018. ISSN 2214-7934. Disponível em: <http://dx.doi.org/10.1007/978-3-319-91086-4_5>.

MARCHAND, H. et al. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, Elsevier BV, v. 123, n. 1–3, p. 397–446, Nov 2002. ISSN 0166-218X. Disponível em: <http://dx.doi.org/10.1016/S0166-218X(01)00348-1>.

MARTELLO, S.; PISINGER, D.; TOTH, P. New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, Elsevier BV, v. 123, n. 2, p. 325–332, Jun 2000. ISSN 0377-2217. Disponível em: <http://dx.doi.org/10.1016/S0377-2217(99)00260-X>.

MATHEWS, G. B. On the partition of numbers. *Proceedings of the London Mathematical Society*, Wiley, s1-28, n. 1, p. 486–490, Nov 1896. ISSN 0024-6115. Disponível em: <http://dx.doi.org/10.1112/plms/s1-28.1.486>.

MAURI, G. R.; RIBEIRO, G. M.; LORENA, L. A. A new mathematical model and a lagrangean decomposition for the point-feature cartographic label placement problem. *Computers & Operations Research*, Elsevier, v. 37, n. 12, p. 2164–2172, 2010.

MITCHELL, J. E. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, Oxford, UK, v. 1, n. 1, p. 65–77, 2002.

MURITIBA, A. E. F. et al. Algorithms for the bin packing problem with conflicts. *Informs Journal on computing*, INFORMS, v. 22, n. 3, p. 401–415, 2010.

NAHAR, S.; SAHNI, S.; SHRAGOWITZ, E. Simulated annealing and combinatorial optimization. In: IEEE. *23rd ACM/IEEE Design Automation Conference*. [S.l.], 1986. p. 293–299.

NOGUEIRA, B.; PINHEIRO, R. G. S.; SUBRAMANIAN, A. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, Springer Science and Business Media LLC, v. 12, n. 3, p. 567–583, Mar 2017. ISSN 1862-4480. Disponível em: <http://dx.doi.org/10.1007/s11590-017-1128-7>.

NOGUEIRA, B.; PINHEIRO, R. G. S.; TAVARES, E. Iterated local search for the generalized independent set problem. *Optimization Letters*, Springer Science and Business Media LLC, v. 15, n. 4, p. 1345–1369, Oct 2020. ISSN 1862-4480. Disponível em: <http://dx.doi.org/10.1007/s11590-020-01643-7>.

PFERSCHY, U.; SCHAUER, J. The knapsack problem with conflict graphs. *Journal of Graph Algorithms and Applications*, Journal of Graph Algorithms and Applications, v. 13, n. 2, p. 233–249, 2009. ISSN 1526-1719. Disponível em: <http://dx.doi.org/10.7155/jgaa.00186>.

PFERSCHY, U.; SCHAUER, J. The maximum flow problem with disjunctive constraints. *Journal of Combinatorial Optimization*, Springer, v. 26, n. 1, p. 109–119, 2013.

PFERSCHY, U.; SCHAUER, J. Approximation of knapsack problems with conflict and forcing graphs. *Journal of Combinatorial Optimization*, Springer Science and Business Media LLC, v. 33, n. 4, p. 1300–1323, Jun 2016. ISSN 1573-2886. Disponível em: <http://dx.doi.org/10.1007/s10878-016-0035-7>.

PISINGER, D. Where are the hard knapsack problems? *Computers & Operations Research*, Elsevier BV, v. 32, n. 9, p. 2271–2284, Sep 2005. ISSN 0305-0548. Disponível em: <http://dx.doi.org/10.1016/j.cor.2004.03.002>.

PISINGER, D.; TOTH, P. Knapsack problems. *Handbook of Combinatorial Optimization*, Springer US, p. 299–428, 1998. Disponível em: <http://dx.doi.org/10.1007/978-1-4613-0303-9_5>.

SADYKOV, R.; VANDERBECK, F. Bin packing with conflicts: a generic branch-and-price algorithm. *INFORMS Journal on Computing*, INFORMS, v. 25, n. 2, p. 244–255, 2013.

SALEM, M. B. et al. Optimization algorithms for the disjunctively constrained knapsack problem. *Soft Computing*, Springer, v. 22, p. 2025–2043, 2018.

SALKIN, H. M.; KLUYVER, C. A. D. The knapsack problem: A survey. *Naval Research Logistics Quarterly*, Wiley, v. 22, n. 1, p. 127–144, Mar 1975. ISSN 1931-9193. Disponível em: <http://dx.doi.org/10.1002/nav.3800220110>.

SHI, H. Solution to 0/1 knapsack problem based on improved ant colony algorithm. In: IEEE. *2006 IEEE International Conference on Information Acquisition*. [S.l.], 2006. p. 1062–1066.

SNOEK, J.; LAROCHELLE, H.; ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, v. 25, 2012.

TARJAN, R. E.; TROJANOWSKI, A. E. Finding a maximum independent set. *SIAM Journal on Computing*, SIAM, v. 6, n. 3, p. 537–546, 1977.

TOTH, P. Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, Springer, v. 25, n. 1, p. 29–45, 1980.

VOSS, S. et al. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Springer US, 1998. ISBN 9781461557753. Disponível em: <http://dx.doi.org/10.1007/978-1-4615-5775-3>.

VOß, S. Meta-heuristics: The state of the art. *Local Search for Planning and Scheduling*, Springer Berlin Heidelberg, p. 1–23, 2001. ISSN 0302-9743. Disponível em: <http://dx.doi.org/10.1007/3-540-45612-0_1>.

WEI, R.; MURRAY, A. T. An integrated approach for addressing geographic uncertainty in spatial optimization. *International Journal of Geographical Information Science*, Taylor & Francis, v. 26, n. 7, p. 1231–1249, 2012.

WEI, Z. et al. Responsive strategic oscillation for solving the disjunctively constrained knapsack problem. *European Journal of Operational Research*, Elsevier, 2023.

YAMADA, T.; KATAOKAT, S.; WATANABET, K. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *IPSJ Journal*, 2002.

YAMADA, T.; TAKEOKA, T. An exact algorithm for the fixed-charge multiple knapsack problem. *European Journal of Operational Research*, v. 192, n. 2, p. 700–705, 2009. ISSN 0377-2217. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0377221707010363>.

YANG, X.-S.; DEB, S.; FONG, S. Metaheuristic algorithms: optimal balance of intensification and diversification. *Applied Mathematics & Information Sciences*, Natural Sciences Publishing Corp, v. 8, n. 3, p. 977, 2014.