Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Um algoritmo Genético para o Problema Bin Packing com Conflitos

Santo André – SP
2025

Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Um algoritmo Genético para o Problema Bin Packing com Conflitos

Projeto de graduação em computação apresentado como requisito para a obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal do ABC.

Aluno: Mateus Flosi Molero

Orientadora: Profa. Dra. Carla Negri Lintzmayer

Santo André – SP
2025

Resumo

Os algoritmos genéticos são muito utilizados para tratar problemas NP-Difíceis, como o *Bin Packing*. Neste problema, temos um conjunto de itens que devem ser alocados no menor número possível de caixas, sem que a capacidade máxima das caixas seja ultrapassada. O *Bin Packing* com Conflitos é uma variação do problema, onde alguns itens não podem ser empacotados em uma mesma caixa. Na literatura, é possível encontrar algumas implementações utilizando algoritmos genéticos, como o GGA-CGT, para a versão original, contudo, aparenta não existir algo semelhante que trata a versão com conflitos. O objetivo deste projeto foi adaptar o GGA-CGT para o *Bin Packing* com Conflitos. Os experimentos realizados mostraram que o algoritmo adaptado foi capaz de obter aproximadamente 60% das soluções ótimas conhecidas, além de melhorar uma instância do conjunto de testes disponível na literatura.

Palavras-Chave: *Bin Packing* com Conflitos, Algoritmo Genético, Otimização Combinatória.

Sumário

1 Introdução

O Problema *Bin Packing* (BPP, do inglês *Bin Packing Problem*), é um problema de otimização combinatória que se enquadra na classe NP-Difícil [10] cujo objetivo é empacotar um conjunto de objetos no menor número possível de recipientes, chamados *bins*, onde cada objeto possui um peso e os *bins* possuem uma capacidade máxima. O *Bin Packing* com Conflitos (BPPC, do inglês *Bin Packing Problem with Conflicts*), adiciona conflitos entre alguns pares de itens, indicando que eles não podem ser empacotados no mesmo *bin*.

Conforme o número de itens aumenta, o espaço de soluções cresce exponencialmente, tornando inviável testar todas as possibilidades em tempo hábil. Além disso, o problema é NP-Difícil, de forma que não há garantias de encontrar um algoritmo que sempre dê uma solução ótima em tempo polinomial a menos que P = NP. Por isso, é comum o uso de heurísticas e meta-heurísticas, que permitem encontrar soluções razoáveis em um tempo de execução significativamente menor. Na literatura, há diversas implementações com esse objetivo, como os algoritmos propostos por Gendreau, Laporte e Semet [13], Maiza e Radjef [17], Bacci e Nicoloso [2], todos voltados para o tratamento do BPPC.

O BPP e suas variantes podem ser encontrados de várias formas na indústria, como, por exemplo, em logística e transporte, alocação de recursos computacionais na nuvem e corte de materiais na manufatura. Em todos esses problemas, uma boa implementação de BPP pode ser utilizada para redução de custos em transporte, armazenamento e desperdícios de materias, o que também impacta iniciativas sustentáveis, além de poder otimizar processos e trazer soluções mais eficientes, trazendo vantagem competitiva no mercado.

San Martin *et al.* [14] publicaram em 2023 uma revisão bibliográfica que analisou aproximadamente 21 trabalhos feitos nos últimos 20 anos voltados ao tratamento do BPP, dividindo esses artigos em três categorias: técnicas heurísticas, meta-heurísticas e exatas. Além de citar brevemente a contribuição de cada trabalho, também foi selecionado o melhor algoritmo de cada categoria, levando em consideração os resultados apresentados nos seus respectivos artigos. A melhor meta-heurística foi apresentada por Quiroz-Castellanos *et al.* [21] em 2015, um algoritmo genético nomeado GGA-CGT.

Curiosamente, não foi encontrado na literatura nenhuma implementação de algoritmo genético que trate o BPPC. Nosso objetivo foi adaptar o GGA-CGT para trabalhar com conflitos, o que gerou uma implementação inédita na literatura que demonstra que novas adaptações do GGA-CGT podem ser feitas para outras versões do BPP.

O restante deste documento está organizado da seguinte forma: a Seção 2 apresenta os principais fundamentos e conceitos que serão utilizados ao longo do documento; a Seção 3 define formalmente o BPPC; a Seção 4 traz um breve levantamento bibliográfico; a Seção 5 explica todas as alterações e adaptações realizadas no GGA-CGT; a Seção 6 discute

os resultados, comparando o desempenho do algoritmo adaptado com outros métodos encontrados na literatura; e, finalmente, a Seção 7 apresenta as principais conclusões do projeto, além de sugerir possíveis melhorias futuras.

2 Fundamentos

Segundo a Teoria da Complexidade Computacional, os problemas podem ser classificados com base na dificuldade de resolvê-los. De modo geral, essas classes são definidas em termos de problemas de **decisão**, ou seja, aqueles cuja resposta é "sim" ou "não" — por exemplo, "é possível empacotar todos os itens em x bins?". A classe **P** inclui os problemas de decisão que podem ser resolvidos por um algoritmo determinístico em tempo polinomial, isto é, cujo tempo de execução cresce de forma polinomial com o tamanho da entrada. Já a classe **NP** (*Nondeterministic Polynomial time*) abrange os problemas de decisão cujas soluções podem ser verificadas em tempo polinomial. Nesse contexto, um conjunto de dados adicional, chamado de **certificado** (por exemplo, um empacotamento válido usando x bins), auxilia na verificação de uma instância cuja solução é "sim".

Um conceito importante na teoria da complexidade é o de **redução**, que consiste em transformar instâncias de um problema em instâncias de outro de modo que a solução do segundo permita resolver o primeiro. A partir disso, define-se que um problema é **NP-Difícil** quando todos os problemas da classe NP podem ser reduzidos a ele em tempo polinomial, o que significa que ele é ao menos tão difícil quanto qualquer problema de NP. Se, além disso, o problema pertence à classe NP, ele é denominado **NP-Completo**.

A implicação prática é que, se fosse encontrada uma solução em tempo polinomial para qualquer problema NP-Completo ou NP-Difícil, todos os problemas da classe NP poderiam ser resolvidos eficientemente, levando à igualdade P = NP. Como até o momento não se sabe resolver problemas NP-Difícieis ou NP-Completos em tempo polinomial e dada a importância prática de muitos deles, é necessário usar algoritmos diferentes. Entre eles, ainda encontram-se os **algoritmos exatos**, que garantem a obtenção da solução ótima, mas são viáveis apenas para instâncias pequenas ou médias devido ao seu elevado custo computacional [23]. Dentre esses algoritmos, pode-se citar o *Branch and Price*, pertencente à família dos métodos de enumeração implícita, além de outros como o *Branch and Bound* e o *Branch and Cut*.

Outra linha de ataque são os **algoritmos de aproximação**, que, embora não garantam uma solução ótima, produzem resultados próximos do ótimo em tempo polinomial, garantindo que o valor obtido esteja dentro de um limite conhecido em relação ao ótimo para qualquer instância. Formalmente, um algoritmo é uma **α-aproximação** se a solução obtida tem um custo de no máximo α vezes o custo ótimo, para problemas de minimização, e de pelo menos $1/\alpha$ vezes o custo ótimo para problemas de maximização [23].

Também se destacam, nesse contexto, as **heurísticas** e **meta-heurísticas**, que cons-

tituem estratégias amplamente empregadas para encontrar soluções em problemas de otimização NP-difíceis, pois conseguem obter boas soluções – e, em alguns casos, até ótimas – com pouco tempo de processamento, principalmente quando comparadas a métodos que exploram exaustivamente todas as soluções possíveis [20].

A principal diferença entre heurísticas e meta-heurísticas está relacionada à sua abrangência. As heurísticas são técnicas mais específicas, desenvolvidas para resolver um problema específico, o que dificulta sua adaptação a outros tipos de problemas. Já as meta-heurísticas são abordagens mais amplas, que funcionam como estruturas flexíveis, podendo ser adaptadas para resolver diferentes problemas [12, 20]. Por exemplo, no trabalho apresentado por San Martin *et al.* [14], são catalogadas pelo menos cinco implementações do algoritmo genético aplicadas ao BPP.

A seguir, será apresentada uma visão geral sobre o funcionamento dos algoritmos genéticos, com base na descrição fornecida por Gendreau e Potvin em *Handbook of Metaheuristics* [12]. Essa abordagem busca sintetizar os principais conceitos, etapas e componentes envolvidos, fornecendo o contexto necessário para compreender como essa meta-heurística é empregada na resolução de problemas complexos.

O **algoritmo genético**, também conhecido como computação evolutiva, algoritmo evolutivo ou programação evolutiva, tornou-se uma meta-heurística amplamente utilizada nos últimos 15 anos para lidar com problemas NP-Difíceis. Na prática, os algoritmos genéticos não diferem muito de outros algoritmos de busca no espaço de soluções, mas utilizam analogias biológicas para explorá-lo de maneira diferenciada.

De forma resumida, armazena-se na memória um conjunto de soluções viáveis, denominado população, que é iterativamente combinada e modificada para gerar novas soluções, ou indivíduos, potencialmente melhores. Essas etapas são chamadas de *crossover* e mutação. Cada indivíduo é avaliado por uma função, mais conhecida como *fitness*, que mede sua qualidade ou aptidão: quanto maior essa qualidade, maiores as chances de o indivíduo se reproduzir; quanto menor, maior a probabilidade de ser descartado da população.

A eficácia do algoritmo genético, no entanto, depende de diversos fatores adicionais, como, por exemplo, a representação das soluções por meio de cromossomos, a geração da população inicial, estratégias de seleção, *crossover* e mutação, bem como a forma de substituição de indivíduos na população e, por fim, as condições de parada do algoritmo.

O **cromossomo**, também chamado de indivíduo no contexto da população do algoritmo, representa uma solução viável para o problema e é composto por genes que codificam as variáveis de decisão. Sua estrutura pode variar conforme a natureza do problema, sendo geralmente classificada como binária ou não binária. Na codificação binária, cada gene assume valores 0 ou 1, como em um problema de alocação de recursos, no qual o cromossomo [0, 1, 1, 0, 1] pode indicar que os recursos 2, 3 e 5 foram ativados. Já em codificações não binárias, os genes podem conter valores inteiros, reais ou estruturas mais complexas, como permutações, listas ordenadas ou até mesmo listas ligadas, úteis

em problemas com estruturas dinâmicas ou tamanho variável. Por exemplo, no problema de alocação de tarefas, um cromossomo como [2, 1, 3, 1, 2] pode indicar que a tarefa 1 será atribuída à máquina 2, a tarefa 2 à máquina 1, e assim por diante.

A **função *fitness*** é responsável por avaliar a aptidão de cada cromossomo dentro da população. Seu papel é atribuir um valor numérico que representa o grau de adaptação ou desempenho da solução em relação ao problema a ser resolvido, podendo estar relacionada com a função objetivo do problema.

A geração da **população inicial** envolve diversas questões, como, por exemplo, o tamanho da população. Uma população muito pequena pode não permitir uma exploração eficaz do espaço de busca, enquanto uma população muito grande pode comprometer o tempo de execução do algoritmo. Outra questão importante é a forma como essa população será gerada. É preferível que isso ocorra de maneira aleatória ou utilizando métodos estatísticos mais sofisticados, com o objetivo de garantir uma maior cobertura do espaço de busca. No entanto, inserir soluções com valores elevados da função *fitness*, geradas por outras heurísticas pode acelerar a convergência do algoritmo genético, contribuindo para a obtenção de soluções de alta qualidade em menos tempo.

É na etapa de **seleção** onde os indivíduos que passarão por *crossover* ou mutação são selecionados. A seleção pode ser realizada de diversas maneiras, mas geralmente leva em consideração a aptidão dos indivíduos. Além de decidir quais indivíduos serão selecionados, também é necessário determinar quantos serão escolhidos, e essa quantidade pode variar conforme as características específicas de cada problema ou cenário. Um método tradicional é o chamado método da roleta, que utiliza uma distribuição de probabilidade proporcional à aptidão de cada indivíduo. Por outro lado, métodos mais elitistas podem abrir mão da aleatoriedade e selecionar diretamente os indivíduos mais ou menos aptos, dependendo da natureza e dos objetivos do problema.

A forma mais tradicional de ***crossover*** consiste na substituição de uma parte da solução de um dos pais pela mesma parte da solução do outro, gerando duas novas soluções viáveis. No entanto, em problemas mais complexos, é comum que apenas um descendente seja gerado. Além disso, em problemas com restrições adicionais, como nos casos do BPP e BPPC, o descendente pode se tornar uma solução inviável. Nesses cenários, torna-se necessário empregar uma lógica mais elaborada para garantir que o *crossover* seja realizado com sucesso e que os descendentes gerados sejam válidos dentro do domínio do problema.

A **mutação**, como o próprio nome indica, é uma pequena alteração em alguns indivíduos. Alguns autores acreditam que a mutação desempenha um papel secundário, sendo utilizada apenas para preservar a diversidade da população. Já outros consideram que o *crossover* seja, na verdade, irrelevante, atribuindo à mutação o papel principal na melhoria das soluções encontradas.

Por ser um algoritmo iterativo, cada rodada de *crossover* e mutação gera uma nova

População; essa rodada é chamada de **geração** no contexto de algoritmos genéticos. Em alguns casos, uma geração pode ser completamente substituída pela seguinte, usando *crossover* e mutação para substituir integralmente a população anterior. No entanto, isso pode significar que as melhores soluções não sejam mantidas, por isso é mais comum que apenas uma parte da população seja substituída. Essa estratégia assegura a sobrevivência dos indivíduos mais aptos até então, mas é preciso tomar cuidado ao escolher quais indivíduos serão substituídos para que a população mantenha sua diversidade.

Sem uma **condição de parada**, o algoritmo genético poderia executar indefinidamente. As condições de parada mais comuns incluem: (i) a definição de um número máximo de gerações; (ii) um tempo máximo de execução; (iii) o monitoramento da diversidade da população, interrompendo a execução quando essa diversidade atinge níveis muito baixos; e (iv) a obtenção de um resultado ótimo, que, em alguns casos específicos, é conhecido previamente ou pode ser calculado por meio de limitantes inferiores no valor de uma solução ótima, que pode ou não ser alcançável, servindo como referência para avaliar a qualidade das soluções encontradas.

Vale salientar que a quantidade de indivíduos selecionados para *crossover* e mutação não precisa ser fixa durante toda a execução do algoritmo. Pode-se utilizar cálculos da diversidade populacional para selecionar um número maior de indivíduos para mutação do que para *crossover*, ou aplicar a mutação apenas em algumas gerações específicas. Inclusive, pode-se adotar uma taxa mais alta de *crossover* nas primeiras gerações, utilizando posteriormente a mutação como principal estratégia para escapar de máximos locais. Existem diversas estratégias, e sua eficácia pode variar conforme o tipo de problema a ser resolvido.

3 Bin Packing com Conflitos

Considere n um inteiro positivo que indica o número de itens disponíveis, w_i reais positivos para $i \in \{1, \ldots, n\}$ que indicam o peso de cada item i e W um número real positivo que indica a capacidade de uma caixa (*bin*). O **Problema *Bin Packing*** (BPP) consiste em distribuir os n itens no menor número de *bins* de forma que a soma dos pesos dos itens em cada *bin* não ultrapasse W.

Martello e Toth [19] apresentaram uma formulação em programação linear inteira para o BPP. Utilizam-se duas variáveis binárias na modelagem: y_k, que indica se o *bin* k está sendo utilizado (quando tem valor 1) ou não (quando tem valor 0), e x_{ik}, que indica se o item i foi alocado no *bin* k (valor 1) ou não (valor 0). Note que o maior número de *bins* que pode-se usar é n, se cada item for alocado em uma *bin* diferente. Assim, a formulação

a seguir descreve o problema:

$$\text{Minimizar} \sum_{k=1}^{n} y_k \tag{1}$$

$$\text{Sujeito a} \sum_{k=1}^{n} x_{ik} = 1 \qquad \forall i \in \{1, \ldots, n\} \tag{2}$$

$$\sum_{i=1}^{n} w_i x_{ik} \leq W y_k \qquad \forall k \in \{1, \ldots, n\} \tag{3}$$

$$y_k \in \{0, 1\} \qquad \forall k \in \{1, \ldots, n\} \tag{4}$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, n\} \tag{5}$$

onde a Equação (1) indica que o objetivo é minimizar o número de *bins* utilizados na solução, a Equação (2) indica que todo item i deve estar em um, e somente um, *bin*, a Equação (3) indica que a soma dos pesos dos itens que estão alocados em um *bin* k não devem superar a sua capacidade máxima e as Equações (4) e (5) indicam o domínio das variáveis utilizadas.

No ***Bin Packing* com Conflitos** (BPPC), tem-se também os n itens, com pesos w_i para cada item $i \in \{1, \ldots, n\}$, capacidade W das *bins* e, além disso, considera ainda que certos pares de itens não podem ser colocados na mesma *bin*. O objetivo é distribuir os n itens no menor número de *bins* de forma que a soma dos pesos dos itens em cada *bin* não ultrapasse W e que itens conflitantes não fiquem na mesma *bin*.

Esses conflitos são geralmente representados por um grafo, no qual os vértices representam os itens e as arestas indicam pares de itens conflitantes que devem ser separados em *bins* diferentes. Um **grafo** $G = (V, E)$ é uma estrutura matemática composta por um conjunto de vértices V e um conjunto de arestas E, frequentemente utilizada para modelar relações par-a-par entre elementos. Um **clique** em um grafo G é um subconjunto de vértices $C \subseteq V$ que induzem um subgrafo completo, isto é, todos os pares de vértices em C estão conectados por uma aresta. No contexto do BPPC, cliques correspondem a conjuntos de itens mutuamente conflitantes, onde cada item precisa ser alocado em uma caixa diferente.

Gendreau, Laporte e Semet [13] criaram uma formulação em programação linear inteira para o BPPC a partir de uma ampliação nas restrições do BPP ao incorporar um grafo de conflitos $G = (V, E)$, no qual cada aresta indica um par de itens que não pode ser alocado no mesmo *bin*. A nova restrição indica que se os itens i e j têm conflito, no máximo um deles pode ir para uma bin k:

$$x_{ik} + x_{jk} \leq 1 \qquad ij \in E, \forall k \in [1, .., n] \tag{6}$$

Tanto o BPP quanto o BPPC são problemas NP-Difíceis [10], o que implica que, a

menos que P = NP, não existe solução em tempo polinomial para esses problemas.

4 Levantamento Bibliográfico

San Martin *et al.* [14], em 2023, analisaram 21 trabalhos feitos nos últimos 20 anos voltados ao BPP, dividindo esses artigos em três categorias: heurísticas, meta-heurística e técnicas exatas. O algoritmo derivado de uma meta-heurística que encontrou mais soluções ótimas nos casos testados foi o algoritmo genético GGA-CGT [21], obtendo uma taxa de sucesso em 99,20% dos casos testados. A heurística CNS_BP [4] obteve uma solução ótima em 99,81% dos casos, contudo sua execução acabou sendo quase duas vezes mais lenta.

Ao contrário do que foi observado para o algoritmo genético, algumas abordagens relacionadas ao CNS_BP foram identificadas na literatura para a versão com conflitos. Um exemplo é o trabalho de Ekici [7], que, embora apresente ideias próximas, trata de uma variação do BPPC, onde os bins não possuem tamanho fixo. Por esse motivo, tal estudo não foi incluído na nossa revisão da literatura.

Apesar de não encontrarmos nenhuma implementação de algoritmo genético para o BPPC na literatura, não é difícil encontrar trabalhos utilizando outras meta-heurísticas ou criando novas heurísticas. A seguir, apresentamos um pequeno resumo dos principais trabalhos existentes na literatura voltados para o desenvolvimento de algoritmos que tratam o BPPC:

1. **An Approximation Scheme for Bin Packing with Conflicts [Klaus Jansen] (1999) [15]:** Adapta um algoritmo de aproximação feito para o BPP clássico, tendo um fator de aproximação $(1 + \varepsilon)$, em que $\varepsilon > 0$.

2. **Heuristics and lower bounds for the bin packing problem with conflicts [Michel Gendreau, Gilbert Laporte, Frédéric Semet] (2004) [13]:** Propõe seis heurísticas e duas formas de calcular um limite inferior para o BPPC, sendo um mais simples, reaproveitando um limite valido para o BPP, e outro com uma lógica mais complexa e precisa. Das seis heurísticas, a melhor é uma que mistura cliques no grafo e no seu complemento com o Fisrt Fit Decreasing.

3. **On Bin Packing with Conflicts [Leah Epstein, Asaf Levin] (2008) [9]:** O artigo propõe vários algoritmos de aproximação baseados em tipos específicos de grafos, como, por exemplo, grafos perfeitos, de intervalo e bipartidos, resolvendo as versões online e offline do problema. O problema é offline quando toda a entrada é conhecida, enquanto que no online a entrada vai aparecendo ao longo do tempo.

4. **Algorithms for the Bin Packing Problem with Conflicts [Albert E. Fernandes Muritiba, Manuel Iori, Enrio Malaguti, Paolo Toth] (2010) [11]:**

São apresentados novos limites superiores e inferiores que, em conjunto com heurísticas clássicas, auxiliam o algoritmo proposto pelo autor, baseado em *Branch and Price*. O autor utiliza 800 instâncias para fazer os testes, obtendo o resultado ótimo em 780 delas.

5. **A new lower bound for bin packing problem with general conflicts graph [Mohamed Maiza, Christelle Gueret] (2010) [18]:** Faz uma pequena alteração no limite inferior proposto por Muritiba *et al*, que calcula um limite inferior por meio de cliques no grafo. A alteração proposta pelo autor é a interatividade na seleção das cliques, ou seja, após a maior clique ser identificada, o grafo é alterado, removendo os vértices e as suas arestas. Com esse novo grafo, é feito o procedimento novamente, identificando a maior clique e assim por diante.

6. **New lower bounds for bin packing problems with conflicts [Ali Khanafer, François Clautiaux, El-Ghazali Talbi] (2010) [16]:** Utiliza funções diferencias e funções diferencias dependentes de dados para calcular um limite inferior. Comparou os seus resultados com os apresentados por Muritiba *et al.*; utilizando as mesmas instâncias, obtendo uma pequena melhora nos resultados.

7. **Heuristics for Solving the Bin-Packing Problem with Conflicts [Mohamed Maiza, Mohammed Said Radjef] (2011) [17]:** Propõe duas novas heurísticas, uma adaptada da versão tradicional do BPP e outra inspirada no mesmo princípio proposto por Gendreau *et al*. Os autores testaram o desempenho do algoritmo proposto em instâncias que aparentemente são diferentes das vistas em Muritiba *et al*; contudo, afirmam que seus algoritmos tiveram um bom desempenho.

8. **A Branch-and-Price Algorithm for the Bin Packing Problem with Conflicts [Samir Elhedhli, Lingzi Li, Marien Gzara, Joe Naoum-Sawaya] (2011) [8]:** Propõem um novo algoritmo *Branch and Price* que utiliza a lógica de cliques no grafo para melhorar os resultados obtidos por Muritiba *et al*. Como resultado, conseguiram resolver de forma ótima 17 instâncias das 20 que ficaram faltando. Contudo, segundo relatado pelos autores, o algoritmo pode demorar um pouco para executar em alguns casos.

9. **Bin Packing with Conflicts: A Generic Branch-and-Price Algorithm [Ruslan Sadykov, François Vanderbeck] (2012) [22]:** É proposto um algoritmo de *Branch and Price* feito utilizando a plataforma BapCod, combinando geração de colunas e heurísticas eficientes, superando resultados anteriores e estabelecendo novos benchmarks em várias instâncias da literatura. Além disso, os autores identificaram que os testes clássicos, como os de Muritiba, não exploravam de forma abrangente todas as características do problema, especialmente em termos de densidade e estrutura dos conflitos. Por esse motivo, criaram novas classes de instâncias, incluindo casos de teste com grafos de conflito gerais, sem restrições estruturais, a fim de

ampliar a diversidade dos cenários experimentais e avaliar o algoritmo de maneira mais completa.

10. **An Improved ACO Algorithm for the Bin Packing Problem with Conflicts Based on Graph Coloring Model [Yuan Ye, Li Yi-jun, Wang Yan-qing] (2014) [24]:** Os autores implementaram uma meta-heurística de colônia de formigas e utilizaram algumas instâncias vistas em Muritiba *et al.*, para testar o desempenho do algoritmo, obtendo resultados próximos, mas não melhores do que os observados na literatura. Segundo os autores, o desempenho é limitado quando o número de itens cresce muito.

11. **A Heuristic Algorithm for the Bin Packing Problem with Conflicts on Interval Graphs [Tiziano Bacci, Sara Nicoloso] (2017) [2]:** A heurística proposta é aplicada a casos em que o grafo de conflitos é de intervalo, ou seja, um grafo cujos vértices podem ser representados por intervalos em uma linha real, havendo uma aresta entre dois vértices sempre que seus intervalos se sobrepõem. Os resultados mostram que o algoritmo supera os concorrentes quando a solução encontrada tem uma média de itens por bin superior a 3,5.

12. **A Study on Exponential-Size Neighborhoods for the Bin Packing Problem with Conflicts [Renatha Capua, Yuri Frota, Luiz Satoru Ochi, Thibaut Vidal] (2017) [5]:** O artigo investiga o uso de *exponential-size neighborhoods*. A proposta consiste em explorar essas vizinhanças complexas por meio de técnicas de programação dinâmica e *branch-and-bound*, permitindo avaliar um número muito maior de movimentos em tempo razoável, superando trabalhos anteriores e atingindo ou superando o benchmark em diversas instâncias.

13. **On the Benchmark Instances for the Bin Packing Problem with Conflicts [Tiziano Bacci, Sara Nicoloso] (2021) [3]:** Nesse artigo, diferente dos demais, os autores demonstram que grafos compostos por uma sequência de vértices conectados em ordem linear são mais fáceis de resolver. Isso ocorre, segundo eles, devido à própria estrutura dos conflitos, o que confirma observações já feitas em outros problemas que também podem ser modelados por grafos. Não por acaso, esse tipo de instância é amplamente adotado como base para testes na literatura, inclusive nos experimentos gerados por Muritiba *et al.* [11].

14. **Approximating Bin Packing with Conflict Graphs via Maximization Techniques [Ilan Doron-Arad, Hadas Shachnai] (2023) [6]:** O artigo prova que o BPPC é mais difícil que o BPP mesmo com grafos de conflitos simples. Mesmo sem fazer testes empíricos, cria um novo framework de aproximação via maximização e, com ele, melhora limites conhecidos para grafos bipartidos, perfeitos ou com partição clique–independente.

15. **Simulated Annealing aplicado ao problema de Bin Packing com Conflitos [Rafael Coelho Monte Alto] (2025) [1]:** O artigo aplica *Simulated Annealing*, uma meta-heurística inspirada no recozimento de metais, ao problema de Bin Packing. Ajustes de parâmetros e vizinhanças influenciam a qualidade e eficiência das soluções, mas o método encontrou a solução ótima em apenas 17,5% das instâncias, mostrando-se uma alternativa competitiva, mas não superior aos benchmarks existentes, como, por exemplo o de Ruslan e Vanderbeck [22].

16. **Exact and approximate size reduction for the bin packing problem with conflicts [Wassim Zahrouni, Hichem Kamoun] (2025) [25]:** Os autores tratam o BPPC via redução no tamanho de instâncias, generalizando critérios de dominância e propondo procedimentos de redução exata e aproximada. A contribuição está em oferecer um pré-processamento que simplifica o problema e melhora limitantes, mas os testes em exemplos ilustrativos não buscaram superar benchmarks. Como limitação, as reduções nem sempre resolvem o problema completo e podem descartar soluções ótimas no caso aproximado.

O GGA-CGT é um algoritmo genético desenvolvido para o BPP. Foi apresentado por Quiroz-Castellanos *et al.* [21], traz muitas contribuições inéditas, além de novas heurísticas em praticamente todas as etapas do algoritmo genético.

Um **cromossomo** ou **indivíduo** é uma solução viável para o BPP. Formalmente, é um conjunto $X = \{B_1, B_2, \ldots, B_{|X|}\}$ em que cada $B_k \subseteq \{1, \ldots, n\}$ é uma *bin*, $B_j \cap B_k = \emptyset$ para todo $j \neq k$ e $\bigcup_{B \in X} = \{1, \ldots, n\}$, sendo ainda necessário que, para cada *bin* B_k, a soma dos pesos dos itens nela contidos não exceda a capacidade W, isto é, $\sum_{i \in B_k} w_i \leq W$. Computacionalmente, é representado por um vetor de listas ligadas em que o tamanho do vetor é a quantidade de *bins* utilizadas e cada entrada do vetor tem uma lista ligada contendo os itens que estão naquela *bin*. Devido às limitações que o BPP impõe, utilizar a codificação tradicional, binária ou não binária, tornaria muito custosa a validação da viabilidade das soluções geradas no *crossover* ou na mutação. Com isso, é comum que o cromossomo possua um número variável de genes.

A Equação (7) a seguir apresenta a **função *fitness*** F utilizada para avaliar cada *bin*, retornando um valor entre 0 e 1, em que quanto mais próximo de 1, mais cheio está o *bin*, sendo 1 o valor correspondente ao *bin* completamente cheio, e quanto mais próximo de 0, mais vazio ele está, com 0 representando um *bin* totalmente vazio. Finalmente, sendo $B \subseteq \{1, \ldots, n\}$ uma das *bins* da solução, o valor do *fitness* de B, $F(B)$, é definido como:

$$F(B) = \sum_{i \in B} \left(\frac{w_i}{W}\right)^2 \quad . \tag{7}$$

Para avaliar um indivíduo $X = \{B_1, \ldots, B_{|X|}\}$ como um todo, calcula-se a média dos

valores de *fitness* de todos os *bins* que compõem a solução por meio da função *fitness* H mostrada a seguir:

$$H(X) = \frac{\sum_{B \in X} F(B)}{|X|} \quad . \tag{8}$$

O valor $H(X)$ também está no intervalo de 0 a 1 e expressa a qualidade global da solução. Quanto mais próximo de 1 for esse valor, maior é o aproveitamento médio dos *bins*, o que indica que há poucos *bins* com espaço ocioso e, portanto, a solução é mais eficiente. Em contraste, valores distantes de 1 revelam que podemos ter alguns *bins* sendo subutilizados, caracterizando um uso ineficiente da capacidade disponível.

A geração da **população inicial** é realizada com um algoritmo *First Fit* adaptado, onde os itens que possuem peso superior a 50% da capacidade do *bin* são alocados primeiro. Em seguida, os itens restantes são distribuídos com base em permutações aleatórias, a fim de gerar soluções variadas. O *First Fit* é uma heurística clássica do BPP, onde cada item é alocado no primeiro *bin* em que sua inclusão não exceda a capacidade.

A **seleção** dos indivíduos que serão utilizados no *crossover* é feita com base em uma porcentagem dos mais bem adaptados. Na mutação, adota-se uma lógica similar, porém são escolhidos os indivíduos com pior desempenho.

Explicar a lógica do ***crossover*** em algoritmos genéticos para o BPP pode ser mais complexo do que parece à primeira vista. Para facilitar a compreensão, Quiroz-Castellanos *et al.* [21] apresentam um exemplo prático no artigo original, que foi traduzido e adaptado a seguir.

Considere uma instância do BPP com $W = 10$, $n = 9$ e pesos $w = (6, 3, 7, 8, 5, 2, 2, 5, 2)$. Cada gene representa um bin e armazena os itens nele alocados. Os *bins* são organizados em ordem decrescente da soma dos pesos dos itens contidos nele. A Figura 1 exibe duas soluções retiradas da população, onde, por exemplo, o *bin* A está armazenando os itens 4 e 6, cujo peso total é 10.

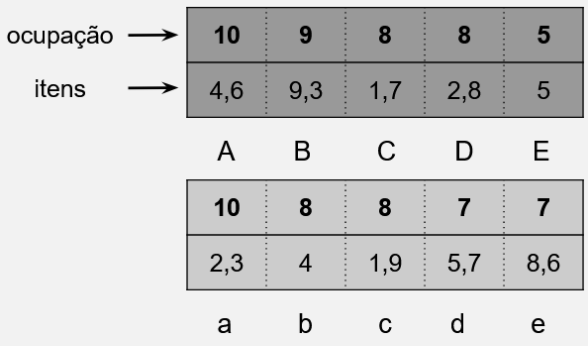

Figura 1: Exemplos de duas soluções retiradas da população. A solução superior usa as bins A,B,C,D,E e a solução inferior usa as bins a,b,c,d,e.

A comparação entre os pais começa pelo primeiro gene, analisando os bins em

paralelo, um a um — por exemplo, bin A com bin a, bin B com bin b, e assim por diante. Para cada par de bins, aquele que estiver mais cheio (isto é, com maior soma dos pesos dos itens) é herdado primeiro pela nova solução. Em seguida, o outro bin do par também é copiado. Se os dois bins tiverem exatamente a mesma capacidade total, dá-se preferência ao bin do primeiro pai. Por exemplo, ao comparar os bins A e a, herdamos primeiro o bin A. A Figura 2 ilustra esse processo com um exemplo prático de comparação e herança entre os bins dos dois pais.

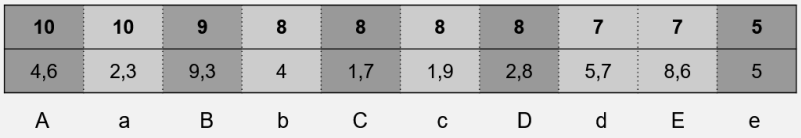

Figura 2: Ordem de herança.

Em seguida, é possível que alguns itens apareçam repetidos na solução gerada, por exemplo, o item 2 pode ter sido incluído tanto no bin B quanto no bin a. Para resolver isso, removemos os bins que contenham itens já presentes em algum bin anterior (como B, b, C, D e e). Após esse ajuste, podem restar itens que ainda não foram alocados, como os itens 7 e 8. A Figura 3 apresenta o estado parcial da solução após essa etapa, com os itens duplicados já removidos e os itens ainda livres para alocação.

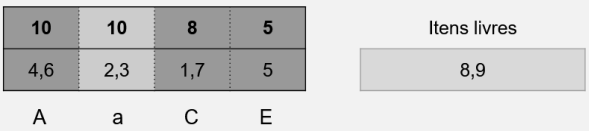

Figura 3: Estado parcial.

Para completar o novo indivíduo (filho), aplicamos a heurística *First Fit* aos itens que ainda não haviam sido alocados. Com isso, os itens livres 8 e 9 foram reinseridos na solução final, sendo alocados, respectivamente, nos bins E e C, como pode ser visualizado na Figura 4.

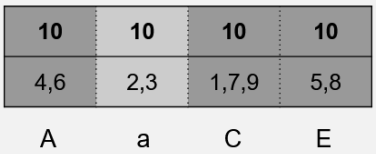

Figura 4: Novo indivíduo.

O algoritmo genético possui alguns parâmetros configuráveis que influenciam diretamente seu desempenho e a qualidade das soluções obtidas. Entre eles, destaca-se a taxa de mutação, que define a quantidade de indivíduos que serão selecionados para sofrer mutação. Além disso, há outras configurações importantes, como a taxa de *crossover*, que determina quantos indivíduos serão gerados por cruzamento, o tamanho da população e o número máximo de gerações, todos ajustáveis conforme as necessidades do problema e

a estratégia adotada.

Quiroz-Castellanos *et al.* [21] também ilustram o funcionamento da **mutação** por meio de um exemplo didático. A seguir, apresentamos a tradução e adaptação desse exemplo para esclarecer como a mutação atua no contexto do BPP.

Considere uma instância do BPP com $W = 10$, $n = 10$ e pesos $w = (6, 3, 7, 8, 4, 2, 2, 5, 2, 8)$. Cada gene representa um bin e armazena os itens nele alocados. Os *bins* são organizados em ordem decrescente da soma dos pesos dos itens contidos nele. A Figura 5 ilustra uma solução selecionada da população.

10	10	9	8	8	4
4,6	2,8	9,3	1,7	10	5
A	B	C	D	E	F

Figura 5: Exemplo de uma solução retirada da população.

A solução possui 6 *bins*, dos quais 4 não estão cheios. Com uma taxa de mudança igual a 3, decide-se remover 2 *bins*. Os menos preenchidos — E e F — são eliminados, e seus itens (10 e 5). A Figura 6 resume essa etapa.

Figura 6: Remoção dos bins menos preenchidos e preparação para o rearranjo dos itens.

Caso fosse aplicada uma heurística tradicional de empacotamento, como *First Fit* (FF) ou *Best Placement* (BP), a solução final seria idêntica à original, pois os bins remanescentes não possuem espaço suficiente.

Suponha, no entanto, que uma heurística de rearranjo por pares permita realocar itens entre os bins. Nesse caso, seria possível alcançar a solução ótima ao substituir o item 1 do bin D pelo item 10. Assim, os itens livres são reinseridos de forma eficiente, resultando na nova configuração apresentada na Figura 7.

10	10	9	10	10
4,6	2,8	9,3	10,7	5,1
A	B	C	D	E

Figura 7: Solução final após o rearranjo.

Os descendentes gerados pelo *crossover* e pela mutação substituem os indivíduos com menor aptidão na população, produzindo uma nova população a cada iteração do algo-

ritmo.

O algoritmo possui duas **condições de parada**: uma quando é encontrada uma solução cujo número de *bins* é igual ao limitante inferior e outra quando o número máximo de gerações é atingido. O limitante inferior no valor da solução ótima utilizado por Quiroz-Castellanos *et al.* [21] não será detalhado no nosso trabalho, uma vez que é específico para o BPP.

5 Nosso algoritmo genético para o BPPC

Nosso algoritmo é uma adaptação do que foi implementado por Quiroz-Castellanos *et al.* [21], cuja qualidade foi reconhecida por San Martin *et al.* [14] como a melhor metaheurística para o BPP. Adaptar uma lógica que já funciona para outro problema, além de ser mais fácil do que implementar algo do zero, pode trazer resultados melhores, especialmente quando se parte de uma solução consolidada na literatura. Vale destacar que, até onde investigamos, não foram encontradas outras implementações de algoritmos genéticos aplicados ao BPPC, o que reforça o caráter inédito desta adaptação.

Apesar das adaptações necessárias para que o algoritmo atendesse às restrições do BPPC, sua estrutura geral conserva a lógica do algoritmo original. As modificações realizadas incluíram a adição dos testes gerados por Muritiba *et al.* [11] e Sadykov e Vanderbeck [22][1], a adaptação do armazenamento dos dados do problema, a verificação de conflitos entre os itens, além de ajustes na geração da população inicial, no *crossover*, na mutação e no cálculo da função de aptidão. O Algoritmo 1 apresenta o pseudocódigo geral da estrutura original do algoritmo base, servindo como referência para contextualização.

Algoritmo 1 Estrutura do algoritmo genético sobre uma instância I de um problema.

```
1: Função AlgoritmoGenetico(I)
2:     S ← GeraPopulacaoInicial(I)
3:     melhor_solucao ← EncontraMelhorSolucao(S)
4:     Enquanto não atingir condição de parada faça
5:         S ← Crossover(S)
6:         S ← Mutacao(S)
7:         melhor_solucao ← EncontraMelhorSolucao(S)
8:     Devolve melhor_solucao
```

Foi necessário ampliar a estrutura de armazenamento dos dados dos itens, que anteriormente consistia em um *array* simples de valores decimais contendo apenas os pesos, para que também armazenasse as relações de conflito existentes. Além disso, no formato dos testes, cada conflito entre itens é informado apenas uma vez e sempre no item de

[1] Encontrados em https://w1.cirrelt.ca/~vidalt/en/research-data.html, onde também estão disponíveis as melhores soluções conhecidas na literatura para cada caso de teste ou *Best Known Solution* (BKS).

menor índice conforme a ordem do problema. Por exemplo, se há um conflito entre o item 2 e o item 5, essa informação aparece somente na descrição do item 2. Para otimizar a verificação de conflitos entre itens, implementou-se um tratamento que garante a presença do conflito em ambos os itens envolvidos.

Com essas alterações realizadas, foram feitos também ajustes nos operadores do algoritmo para que a geração da população inicial, o *crossover* e a mutação considerassem os conflitos entre os itens, garantindo que as soluções se mantivessem viáveis ao longo de toda a execução.

No caso da população inicial e do *crossover*, foram incluídas validações para verificar se a adição de um novo item a um *bin* causaria conflitos com os itens já presentes, além da tradicional verificação de sobrepeso, conforme ilustrado no Algoritmo 2.

Algoritmo 2 Validação de adição de um item i em uma bin B.

Função ValidaAdicao($I = (n, w, W, G)$, B, i)
  Se $\sum_{j \in B} w_j + w_i > W$ **então** ▷*o bin não pode ficar com sobrepeso*
    Devolve `Falso`
  Para cada $j \in B$ **faça** ▷*itens de B não podem conflitar com i*
    Se $ij \in E(G)$ **então**
      Devolve `Falso`
  Devolve `Verdadeiro`

Já para a mutação, é permitido que certos itens sejam removidos de um *bin* para viabilizar a entrada de outros. Por esse motivo, durante as trocas, que podem ser simples, quando dois itens são removidos de um *bin* para a adição de apenas um, ou duplas, quando dois itens são removidos e outros dois são adicionados, realizam-se validações de conflito e capacidade. Essas validações desconsideram temporariamente os itens que seriam removidos, com o objetivo de verificar se a alteração produz uma solução viável. Caso o resultado seja positivo, a modificação é aplicada, resultando em ganhos de qualidade. A lógica implementada para a troca simples encontra-se no Algoritmo 3, enquanto o Algoritmo 4 apresenta a lógica da troca dupla.

Algoritmo 3 Validação de troca simples de itens k e ℓ por i.

Função ValidaTrocaSimples($I = (n, w, W, G)$, B, i, k, ℓ)
  Se $\sum_{j \in B \setminus \{k,\ell\}} w_j + w_i > W$ **então** ▷*o bin não pode ficar com sobrepeso*
    Devolve `Falso`
  Se $w_i \leq w_k + w_\ell$ **então** ▷*evitar substituições sem ganho efetivo de espaço*
    Devolve `Falso`
  Para cada $j \in B \setminus \{k, \ell\}$ **faça** ▷*itens de $B \setminus \{k, \ell\}$ não podem conflitar com i*
    Se $ij \in E(G)$ **então**
      Devolve `Falso`
  Devolve `Verdadeiro`

Algoritmo 4 Validação de troca dupla de itens k e ℓ por itens i e j.

Função ValidaTrocaDupla($I = (n, w, W, G)$, B, i, j, k, ℓ)
  Se $\sum_{h \in B \setminus \{k,\ell\}} w_h + w_i + w_j > W$ **então** ▷*o bin não pode ficar com sobrepeso*
    Devolve `Falso`
  Se $w_i + w_j \leq w_k + w_\ell$ **então** ▷*evitar substituições sem ganho efetivo de espaço*
    Devolve `Falso`
  Se $ij \in E(G)$ **então** ▷*i e j não podem conflitar*
    Devolve `Falso`
  Para cada $h \in B \setminus \{k, \ell\}$ **faça** ▷*itens de $B \setminus \{k, \ell\}$ não podem conflitar com i ou j*
    Se $ih \in E(G)$ ou $jh \in E(G)$ **então**
      Devolve `Falso`
  Devolve `Verdadeiro`

O cálculo do limitante inferior no número de bins de uma solução foi alterado para utilizar a mesma fórmula vista em Alto [1], mostrada na equação a seguir:

$$LB = \frac{\sum_{i=1}^{n} w_i}{W} \tag{9}$$

Essa equação soma o peso de todos os itens e divide o resultado pela capacidade máxima de cada *bin*. Assim, o valor obtido representa, na melhor das hipóteses, o número mínimo de bins necessário para acomodar todos os itens. No entanto, esse limite pode ser impossível de atingir em certas instâncias, pois admite que todos os *bins*, exceto possivelmente o último, o que pode não ser viável dependendo da natureza do problema.

Além disso, o critério de parada do algoritmo foi modificado: a execução é interrompida quando a melhor solução não apresenta evolução após 25 gerações consecutivas. Caso o limitante inferior não seja encontrado e a solução continue evoluindo, o processamento termina após 100 gerações.

A versão original da criação da população inicial baseia-se em uma adaptação do algoritmo *First Fit*. Nessa adaptação, um item cujo peso é superior à metade da capacidade do *bin* é alocado individualmente, pois combinar dois desses itens em um mesmo *bin* resultaria em sobrepeso. Para garantir a diversidade populacional, os demais itens são distribuídos de forma aleatória, seguindo o comportamento padrão do *First Fit*, no qual cada item é inserido no primeiro *bin* disponível que não viole as restrições do problema. O pseudocódigo correspondente é apresentado no Algoritmo 5.

Algoritmo 5 Geração de uma solução inicial para o BPP, conforme descrito no artigo de Quiroz-Castellanos *et al.*

Função GeraSolucaoInicialBP($I = (n, w, W)$)
  $O \leftarrow \{i \in \{1, \ldots, n\} : w_i > W/2\}$ ▷*itens com peso grande*
  $P \leftarrow \{i \in \{1, \ldots, n\} : w_i \leq W/2\}$ ▷*itens com peso pequeno*
  $S \leftarrow \emptyset$ ▷*a solução é um conjunto de conjuntos de itens (bins)*
  Para cada $i \in O$ **faça** ▷*cria-se um bin novo para cada item grande*
    $S \leftarrow S \cup \{\{i\}\}$
  Para cada $i \in P$ **em ordem aleatória faça** ▷*inserir itens pequenos na solução atual por First Fit*
    $S \leftarrow$ FirstFit(I, S, i)
  Devolve S

Como não encontramos nenhuma implementação de algoritmo genético no BPPC na literatura, consequentemente não temos nenhum algoritmo específico para geração da população inicial, então propusemos uma adaptação. Essa adaptação mantém a estrutura original do algoritmo, mas introduz alterações pontuais para aumentar as chances dos itens mais conflitantes serem alocados em *bins* mais preenchidos.

Primeiramente, os itens cujos pesos ultrapassam a metade da capacidade do *bin* são ordenados de acordo com o número de conflitos, de modo que o item mais conflituoso esteja no primeiro *bin*. Essa estratégia permite que os *bins* que contêm itens mais problemáticos sejam analisados primeiro durante a inserção dos demais itens seguindo a lógica do *First Fit*.

Em seguida, o mesmo princípio é aplicado aos itens restantes, que também são ordenados pelo número de conflitos. No entanto, para preservar a diversidade populacional, esses itens são inseridos de forma parcialmente aleatória. Essa aleatoriedade é controlada por blocos de tamanho dinâmico, que definem o número de itens considerados a cada iteração do *First Fit*. O tamanho dos blocos cresce de maneira iterativa: inicialmente, é gerado um indivíduo com blocos de tamanho 1, seguido por quatro indivíduos com blocos de tamanho 2, e assim sucessivamente, até que toda a população seja construída.

Chamamos essa heurística de *First Fit Decreasing for Conflicts* (FFDC). A seguir, é apresentado um exemplo ilustrativo de seu funcionamento, e o seu pseudocódigo pode ser visto no Algoritmo 6.

Considere uma instância do BPPC com $W = 10$, $n = 10$ e blocos de tamanho 4. A Figura 8 ilustra os pesos e os conflitos de cada item.

index →	1	2	3	4	5	6	7	8	9	10
peso →	3	2	5	7	3	1	6	2	5	1
conflitos →		5, 7, 9		5, 6, 7	2, 4	4, 7, 8, 9	2, 4, 6, 10	6	2, 6	7

Figura 8: Pesos e relações de conflito entre os itens da instância considerada.

Inicialmente, os itens são divididos em dois grupos: o primeiro contém aqueles com peso superior a $W/2$, enquanto o segundo reúne os demais. Em seguida, cada grupo é ordenado de acordo com o número de conflitos associados a cada item. A Figura 9 ilustra os grupos após a ordenação.

7	**4**
6	7
2, 4, 6, 10	5, 6, 7

6	**2**	**5**	**9**	**8**	**10**	**1**	**3**
1	2	3	5	2	1	3	5
4, 7, 8, 9	5, 7, 9	2, 4	2, 6	6	7		

Figura 9: Agrupamento e ordenação dos itens conforme peso e número de conflitos.

Após essa separação e ordenação, os itens do primeiro grupo são alocados individualmente em *bins*. Já os itens do segundo grupo são divididos em subgrupos de quatro elementos, seguindo a ordem apresentada na Figura 9: os quatro primeiros (6, 2, 5, 9) formam o primeiro subgrupo, e os quatro últimos (8, 10, 1, 3) compõem o segundo. A Figura 10 ilustra o resultado dessa etapa.

	A	B
ocupação →	**6**	**7**
itens →	7	4
conflitos →	2, 4, 6, 10	5, 6, 7

6	**2**	**5**	**9**
1	2	3	5
4, 7, 8, 9	5, 7, 9	2, 4	2, 6

8	**10**	**1**	**3**
2	1	3	5
6	7		

Figura 10: Primeira alocação nos bins e formação dos subgrupos.

O primeiro subgrupo teve sua ordem embaralhada aleatoriamente, resultando na sequência (5, 6, 9, 2). Em seguida, seus itens são inseridos nos *bins* seguindo a lógica do *First Fit*, isto é, cada item é alocado no primeiro *bin* disponível que não viole as restrições de peso ou conflito entre itens. A Figura 11 apresenta a disposição dos *bins* após essa etapa.

9	**9**	**5**	**1**
5, 7	2, 4	9	6
2, 4, 6, 10	5, 6, 7, 9	2, 6	4, 7, 8, 9
A	B	C	D

Figura 11: Distribuição dos itens do primeiro subgrupo nos bins segundo o critério First Fit.

Por fim, o mesmo procedimento é aplicado ao segundo subgrupo, que também tem sua ordem embaralhada aleatoriamente, resultando na sequência (10, 1, 8, 3). Seus

itens são então inseridos nos *bins* utilizando a mesma lógica do primeiro subgrupo. A Figura 12 apresenta a solução final obtida.

9	10	10	6
5, 7	2, 4, 10	1, 8, 9	3, 6
2, 4, 6, 10	5, 6, 7, 9	2, 6	4, 7, 8, 9
A	B	C	D

Figura 12: Solução final após a inserção dos itens do segundo subgrupo.

Algoritmo 6 Geração de uma solução inicial para o BPPC, considerando blocos de tamanho b

Função GERASOLUCAOINICIALBPPC($I = (n, w, W, G)$, b)
  $O \leftarrow \{i \in \{1, \ldots, n\} : w_i > W/2\}$ ▷*itens com peso grande*
  $O_{\text{sort}} \leftarrow$ sequência dos itens de O em ordem decrescente do grau no grafo G ▷*o grau indica o número de conflitos*
  $P \leftarrow \{i \in \{1, \ldots, n\} : w_i \leq W/2\}$ ▷*itens com peso pequeno*
  $P_{\text{sort}} \leftarrow$ sequência dos itens de P em ordem decrescente do grau no grafo G
  $Q \leftarrow$ partição de P_{sort} em subconjuntos consecutivos de tamanho b
  $S \leftarrow \emptyset$ ▷*a solução é um conjunto de conjuntos de itens (bins)*
  Para cada $i \in O_{\text{sort}}$ **faça** ▷*cria-se um bin novo para cada item grande*
    $S \leftarrow S \cup \{\{i\}\}$
  Para cada $P' \in Q$ **faça** ▷*inserir itens pequenos na solução atual por First Fit*
    Para cada $i \in P'$**em ordem aleatória faça**
      FIRSTFIT(I, S, i)
  Devolve S

Os **conflitos externos** de um *bin* correspondem ao conjunto de itens que possuem conflito com ao menos um item pertencente ao *bin* $B \subseteq \{1, ..., n\}$. Caso um mesmo item conflite com mais de um item de B, ele é contabilizado apenas uma vez. Denotado por $N(B)$, definimos da seguinte forma:

$$N(B) = \{u \notin B : \exists\, v \in B \text{ tal que } \{uv\} \in E\}. \tag{10}$$

Com base nesse conceito, foi criada uma nova **função *fitness*** que considera os conflitos externos. Essa função também vale entre 0 e 1, em que valores próximos de 1 indicam *bins* com muitos conflitos externos, enquanto valores próximos de 0 indicam poucos conflitos. Para um dado *bin* B, denotamos por $F'(B)$ seu $fitness$ definido da seguinte forma:

$$F'(B) = \frac{|N(B)|}{n - |B|}\ . \tag{11}$$

Durante os testes, as Equações (7) e (11) foram aplicadas a cada *bin*, sendo adotado como *fitness* o maior valor entre elas. Assim, a avaliação favorecia *bins* bem preenchidos ou

com grande quantidade de conflitos externos. No entanto, recalcular a aptidão levando em conta todos os conflitos aumentava significativamente o tempo de execução, especialmente em instâncias com muitas restrições. Por isso, durante a execução do algoritmo genético, optou-se por utilizar apenas a Equação (7). Ao término da execução, a melhor solução é reavaliada considerando novamente o maior valor entre as duas funções, garantindo uma avaliação mais coerente da qualidade da solução sem comprometer o desempenho do algoritmo.

6 Resultados e discussões

Os experimentos foram realizados em uma máquina virtual executada via VMware Workstation 17 Player, com o sistema operacional Ubuntu 22.04.2 Desktop (64 bits). A máquina foi configurada com 8 GB de memória RAM e 4 núcleos de processamento, oferecendo um ambiente controlado e estável para a realização dos testes. O algoritmo, originalmente implementado em C, foi mantido nessa linguagem com o objetivo de preservar a estrutura e o desempenho da versão base utilizada como referência.

Com base nos trabalhos de Capua *et al.* [5] e Alto [1], os resultados serão apresentados sem comparação direta com algoritmos específicos, mas em relação a uma base de dados disponível em `https://w1.cirrelt.ca/~vidalt/en/research-data.html`. Nessa base, o pesquisador Thibaut Vidal disponibiliza tanto as instâncias de teste para o BPPC quanto os melhores resultados obtidos até o momento, consolidando o desempenho de vários algoritmos. Dessa forma, os resultados do GGA-CGT adaptado serão comparados com os melhores valores conhecidos na literatura, também denominados *Best Known Solutions* (BKS), possibilitando uma análise mais objetiva e concisa.

O código-fonte, os casos de teste e os resultados gerados, que demonstram o desempenho e a eficácia do algoritmo implementado, estão todos disponíveis no repositório do projeto no GitHub: `https://github.com/mateusflosi/GGA-CGT-NEW`.

Foram utilizados todos os casos de testes disponilizados por Thibaut Vidal, totalizando 2240 casos de teste, que podem ser divididas em seis classes que estão descritas na Tabela 1.

Tabela 1: Características das 6 classes de instâncias.

Classe	**Casos**	**Fonte**	**Núm. n de itens**	**Capac. W do bin**
U	400	Muritiba *et al.* [11]	120, 250, 500, 1000	150
T	400	Muritiba *et al.* [11]	60, 120, 249, 501	1000
D	360	Sadykov e Vanderbeck [22]	120, 250, 500, 1000	10000
UA	360	Sadykov e Vanderbeck [22]	120, 250, 500, 1000	1000
TA	360	Sadykov e Vanderbeck [22]	60, 120, 249, 501	1000
DA	360	Sadykov e Vanderbeck [22]	120, 250, 500, 1000	10000

Cada classe possui uma lógica diferente para a geração dos pesos dos itens:

- **Classe U (uniforme):** os pesos são sorteados uniformemente no intervalo $[20, 100]$, com capacidade do bin $W = 150$. Assim, cada *bin* acomoda em média de 1 a 3 itens.
- **Classe T (triplets):** os pesos são sorteados uniformemente no intervalo $[150, 200]$, com capacidade do *bin* $W = 1000$. Os itens são gerados em trios de forma que cada tripla complete exatamente a capacidade do *bin*, forçando soluções ótimas com $n/3$ *bins* com o espaço total preenchido.
- **Classe D (dense):** os pesos são sorteados uniformemente no intervalo $[500, 1500]$, com $W = 10000$, permitindo que, em média, cerca de 8 itens caibam em cada *bin*.

As variações UA, TA e DA compartilham a mesma lógica de geração de itens das classes originais, contudo possuem um algoritmo diferente para construir os conflitos. Para as classes U, T e D, os conflitos são gerados de forma simétrica, onde cada item recebe um valor $\alpha_i \in [0, 1]$, e dois itens i e j entram em conflito se $(\alpha_i + \alpha_j)/2 \leq \delta$, para um dado limiar δ. Dessa forma, quanto menor o valor de δ maior será a densidade do grafo de conflitos. Já nas classes UA, TA e DA, o grafo inicia vazio e, em seguida, arestas são inseridas aleatoriamente entre pares de itens até que a densidade de conflitos atinja o valor desejado. Como resultado, obtém-se grafos não intervalares, sem qualquer estrutura pré-definida.

Para avaliar os resultados do experimento vamos analisa-los utilizando as seguintes métricas:

- **#opt.:** número de instâncias onde o BKS foi atingido.
- ***gap* (%):** diferença percentual média entre a solução encontrada em relação ao BKS.
- **T (s):** tempo total de execução em segundos.
- **INO (Instâncias não Otimizadas):** número de instâncias em que o AG não conseguiu evoluir a melhor solução encontrada na população inicial, a não ser que esta atinja o BKS.
- **OM (%, Otimização Média):** melhoria média, em porcentagem, da melhor solução encontrada ao final do AG em relação à solução encontrada na população inicial.
- **Fitness:** valor médio da função de avaliação do algoritmo.

A Tabela 2 compara os resultados de algumas métricas entre o GGA-CGT e o FFDC, considerando a execução com blocos de tamanho 1. Essa visualização permite uma análise direta entre a heurística utilizada na geração da população inicial e o AG, destacando as melhorias obtidas com a aplicação do processo evolutivo.

Tabela 2: Resultado comparativo entre o GGA-CGT e o FFDC.

Classe	Casos	GGA-CGT			FFDC		
		#opt.	***gap* (%)**	***T* (s)**	**#opt.**	***gap* (%)**	***T* (s)**
U	400	280	0.39	889.55	92	1.79	3.92
T	400	289	0.76	521.17	212	5.07	1.45
D	360	358	0.00	898.62	341	0.08	4.04
UA	360	162	1.39	1351.33	0	7.64	1.95
TA	360	170	1.38	380.67	0	11.71	0.94
DA	360	78	9.83	3673.98	7	17.23	2.46
total	2240	1337	2.23	7715.31	652	7.12	14.76

Como esperado, a heurística é executada em um tempo significativamente menor que o AG. No entanto, o número de soluções ótimas encontradas cai quase pela metade e o gap médio é mais de três vezes maior. Assim, sua utilização é recomendada apenas quando se necessita de uma solução muito rápida. Caso seja possível aguardar um pouco mais, vale a pena optar pela execução do AG. A partir deste ponto, consideraremos apenas os resultados do GGA-CGT para as análises seguintes.

A Tabela 3 apresenta os resultados agrupados por classe, incluindo os valores consolidados, permitindo uma visão global do desempenho do algoritmo. Embora semelhante à tabela anterior, conta com a adição novas métricas relacionadas à atuação do AG, possibilitando análises e conclusões adicionais.

Tabela 3: Resultados agrupados por classe.

Classe	Casos	#opt.	*gap* (%)	*T* (s)	INO	OM (%)	Fitness
U	400	280	0.39	889.55	20	0.91	0.9385
T	400	289	0.76	521.17	4	3.31	0.8912
D	360	358	0.00	898.62	2	0.04	0.7719
UA	360	162	1.39	1351.33	1	4.93	0.9841
TA	360	170	1.38	380.67	4	6.81	0.9666
DA	360	78	9.83	3673.98	75	3.14	0.9710
total	2240	1337	2.23	7715.31	106	3.15	0.9203

O algoritmo apresentou desempenho satisfatório em boa parte dos casos testados, atingindo o BKS em aproximadamente 60% das instâncias, com um *gap* médio de 2,23%. É relevante notar que as classes com conflitos simétricos (U, T e D) foram resolvidas de forma mais eficiente, especialmente a classe D, na qual foram encontradas 358 das 360 soluções ótimas. Em contrapartida, o desempenho foi consideravelmente inferior nas classes com conflitos gerados aleatoriamente (UA, TA e DA), onde a taxa de BKS caiu quase pela metade e o *gap* aumentou de forma considerável, em especial na classe DA, que apresentou um *gap* próximo de 10%.

Apesar do padrão observado, as classes aleatórias obtiveram valores mais elevados de *Fitness* e OM, indicando que o algoritmo conseguiu melhorar mais essas instâncias ao longo das iterações. Contudo, é possível notar um contraste interessante na classe d, que embora tenha apresentado os melhores resultados em termos de BKS, registrou os

menores valores de *Fitness* e OM, o que indica que muitas soluções já estavam presentes na população inicial, reduzindo a margem de atuação do AG.

Por outro lado, a análise da métrica INO reforça a menor efetividade da classe DA, que concentrou aproximadamente 70% das instâncias que não foram aprimoradas. Por esse motivo, os esforços para melhorar o algoritmo devem priorizar essaa, dada sua combinação de maior *gap*, elevado número de instâncias não otimizadas e menor quantidade de soluções ótimas encontradas.

Bacci e Nicoloso [3] já indicaram que as classes U,T e D são mais simples, o que ajuda a explicar por que o algoritmo encontra soluções boas logo na geração da população inicial, reduzindo a ação do AG, como evidenciado pelos baixos valores de OM. Além disso, a estratégia utilizada para gerar a população inicial pode ter sido favorecida pela estrutura simétrica de conflitos dessas classes.

No entanto, essa mesma lógica que beneficiou as classes simétricas pode ter prejudicado o desempenho nas classes aleatórias. A utilização de apenas uma heurística na geração da população inicial pode ter limitado a diversidade da população. Isso ajuda a compreender por que, embora o AG tenha atuado de maneira eficaz nessas classes, como indicado pelos altos valores de *Fitness* e OM, os resultados finais ainda apresentaram maiores *gaps* e menor número de soluções ótimas encontradas.

Diante disso, uma melhoria promissora para o GGA-CGT seria implementar uma versão em que a população inicial não seja gerada de forma única, mas construída a partir de diferentes heurísticas. Essa abordagem pode aumentar a diversidade da população e, consequentemente, melhorar o desempenho do algoritmo, em especial para as classes com conflitos aleatórios.

A Tabela 4 apresenta os resultados agrupados por classe e pelo número de itens, permitindo uma análise mais detalhada do comportamento do algoritmo em diferentes tamanhos de problema.

Tabela 4: Resultados consolidados do GGA-CGT

Classe	n	Casos	#opt.	*gap* (%)	T (s)	INO	OM (%)	Fitness
U	120	100	92	0.19	14.83	2	0.95	0.9221
	250	100	72	0.41	53.39	3	0.99	0.9420
	500	100	60	0.48	180.75	6	0.93	0.9453
	1000	100	56	0.50	640.58	9	0.77	0.9444
T	60	100	97	0.13	8.45	1	3.29	0.8814
	120	100	76	0.65	22.39	0	3.38	0.8937
	249	100	58	0.75	90.84	1	3.72	0.9032
	501	100	58	1.51	399.49	2	2.84	0.8867
D	120	90	90	0.00	15.92	0	0.00	0.7646
	250	90	90	0.00	51.27	0	0.06	0.7648
	500	90	90	0.00	174.47	0	0.06	0.7790
	1000	90	88	0.01	656.96	2	0.06	0.7791
UA	120	90	58	1.48	14.95	1	5.51	0.9698
	250	90	41	1.62	62.60	0	5.13	0.9832
	500	90	35	1.34	258.19	0	4.80	0.9907
	1000	90	28	1.11	1015.60	0	4.27	0.9929
TA	60	90	74	0.85	8.96	3	4.82	0.9281
	120	90	47	1.97	21.47	1	6.42	0.9641
	249	90	27	1.76	68.95	0	7.72	0.9830
	501	90	22	0.94	281.29	0	8.28	0.9912
DA	120	90	26	8.47	34.62	15	3.84	0.9547
	250	90	32	10.18	145.41	25	3.00	0.9654
	500	90	20	10.60	625.37	20	2.90	0.9777
	1000	90	15	10.06	2868.57	15	2.80	0.9862

De modo geral, o número de soluções ótimas encontradas diminui à medida que o número de itens cresce, o que evidencia a maior complexidade das instâncias maiores. A única exceção aparece na classe DA, onde o número de soluções ótimas com 250 itens superou o obtido com 120 itens.

O *gap* médio também acompanha esse padrão, aumentando com o número de itens. Com exceções na classe TA, que apresentou um pico nas instâncias com 120 e 249 itens, e na classe DA, onde o pico aparece em 249 itens.

O tempo de execução pode apresentar crescimento exponencial à medida que o número de itens aumenta, já que instâncias maiores tendem a exigir mais *bins*, o que torna as operações de *crossover* e mutação mais custosas. Entretanto, esse crescimento do tempo de processamento pode inviabilizar o uso do algoritmo em cenários com muitos itens, especialmente em ambientes que precisam de resultados rápidos.

Outro indício de baixa diversidade populacional aparece na classe TA, onde tanto o OM quanto o *Fitness* evoluem de forma considerável com o aumento de itens. Contudo, mesmo com essa boa atuação, observou-se uma queda expressiva no número de soluções ótimas encontradas, o que sugere que a população inicial possuia baixa diversidade, o que limitou a exploração do espaço de busca.

A Tabela 5 apresenta os resultados agrupados por classe e pela densidade ρ de conflitos, permitindo uma análise mais detalhada do comportamento do algoritmo para diferentes densidades dos conflitos.

Tabela 5: Resultados consolidados do GGA-CGT

Classe	ρ	**Casos**	**#opt.**	***gap*** **(%)**	***T*** **(s)**	**INO**	**OM (%)**	**Fitness**
U	0	40	38	0.037	26.59	0	1.219	0.9895
	10	40	39	0.012	27.76	0	1.390	0.9900
	20	40	22	0.204	74.14	0	1.700	0.9864
	30	40	9	0.923	95.69	0	2.122	0.9742
	40	40	12	2.159	86.23	6	1.628	0.8958
	50	40	29	0.235	74.67	3	0.339	0.8019
	60	40	29	0.136	95.67	3	0.347	0.8725
	70	40	31	0.147	115.58	3	0.243	0.9274
	80	40	33	0.079	137.90	4	0.075	0.9632
	90	40	38	0.006	155.31	1	0.042	0.9837
T	0	40	13	2.331	116.12	0	10.396	0.9589
	10	40	13	2.049	86.68	0	9.642	0.9592
	20	40	12	1.370	70.92	0	7.650	0.9558
	30	40	17	1.685	56.54	0	4.903	0.9283
	40	40	38	0.057	19.56	0	0.337	0.7565
	50	40	39	0.011	23.69	1	0.128	0.7131
	60	40	38	0.090	29.19	2	0.000	0.8216
	70	40	39	0.007	34.43	1	0.029	0.8970
	80	40	40	0.000	39.49	0	0.000	0.9464
	90	40	40	0.000	44.53	0	0.000	0.9754
D	10	40	38	0.034	49.85	2	0.389	0.9765
	20	40	40	0.000	49.82	0	0.000	0.7259
	30	40	40	0.000	64.99	0	0.000	0.5077
	40	40	40	0.000	78.86	0	0.000	0.5055
	50	40	40	0.000	93.81	0	0.000	0.6282
	60	40	40	0.000	113.06	0	0.000	0.7821
	70	40	40	0.000	131.80	0	0.000	0.8886
	80	40	40	0.000	150.10	0	0.000	0.9500
	90	40	40	0.000	166.32	0	0.000	0.9825
UA	10	40	36	0.093	33.96	0	3.669	0.9896
	20	40	37	0.062	34.96	0	4.189	0.9878
	30	40	34	0.087	55.62	0	4.406	0.9880
	40	40	25	0.197	99.90	0	4.817	0.9871
	50	40	17	0.348	145.76	0	5.001	0.9864
	60	40	10	0.706	197.99	0	5.417	0.9840
	70	40	3	1.355	241.86	0	5.732	0.9788
	80	40	0	2.567	304.82	0	6.337	0.9752
	90	40	0	7.074	236.46	1	4.780	0.9805
TA	10	40	29	0.560	34.87	0	6.636	0.9709
	20	40	13	0.983	35.22	0	5.585	0.9603
	30	40	12	0.951	33.85	0	5.945	0.9596
	40	40	21	0.421	33.66	0	6.288	0.9597
	50	40	21	0.421	34.80	0	6.505	0.9632
	60	40	36	0.060	41.11	0	7.241	0.9686
	70	40	28	0.716	56.27	0	8.307	0.9710
	80	40	6	2.303	50.84	0	8.165	0.9703
	90	40	4	5.988	60.05	4	6.644	0.9760
DA	10	40	40	-0.016	74.58	0	1.595	0.9790
	20	40	30	0.491	327.06	4	1.672	0.9725
	30	40	8	1.913	839.80	17	0.872	0.9540
	40	40	0	4.564	515.22	31	0.468	0.9416
	50	40	0	8.533	462.65	21	1.510	0.9612
	60	40	0	14.241	497.00	2	3.205	0.9747
	70	40	0	19.131	471.69	0	5.743	0.9819
	80	40	0	20.937	290.20	0	7.142	0.9858
	90	40	0	18.637	195.78	0	6.009	0.9884

A diversidade de comportamentos entre classes e densidades torna difícil extrair padrões globais. Por esse motivo, a análise será apresentada de modo objetivo, destacando os pontos mais relevantes observados para cada classe:

- **Classe U:** O tempo de execução tende a crescer conforme aumenta a densidade de conflitos. O número de soluções ótimas se mantém relativamente constante, exceto por uma queda observada para ρ entre 20 e 40, justamente o intervalo em que o algoritmo evoluiu mais a solução inicial, como evidenciado pelo pico de OM. Após esse ponto, o OM diminui consideravelmente, indicando melhora na população inicial e, consequentemente, menor margem de melhoria nas soluções.

- **Classe T:** Apresenta desempenho inferior até $\rho = 30$, mas melhora significativamente nas densidades mais altas, conforme observado no número de soluções ótimas encontradas. Curiosamente, OM e Fitness seguem tendência inversa, enquanto os poucos casos de INO ocorrem em densidades onde todas as outras instâncias foram otimizadas. O tempo de execução, por sua vez, diminui à medida que a densidade de conflitos aumenta.

- **Classe D:** O comportamento é consistente com o observado nas análises anteriores. Os dois casos de INO ocorrem em densidades menores, reforçando que, nesse grupo, as soluções de alta qualidade já estão presentes na população inicial.

- **Classe UA:** O tempo de execução cresce com a densidade de conflitos. O desempenho inicial é bom, mas apresenta queda a partir de $\rho = 40$. O *gap* aumenta gradualmente com o aumento da densidade, atingindo valores elevados a partir de $\rho = 70$. OM cresce com a densidade, enquanto Fitness se mantém praticamente constante, indicando que o algoritmo continua a evoluir as soluções mesmo quando o número de ótimos diminui.

- **Classe TA:** Apresenta comportamento semelhante à classe U: bom desempenho inicial, queda em densidades intermediárias e recuperação nas densidades finais. No entanto, os piores resultados ocorrem em $\rho = 80$ e 90. O tempo de execução é quase constante, com leve aumento a partir de $\rho = 60$. OM e Fitness permanecem praticamente estáveis, sugerindo atuação consistente do algoritmo.

- **Classe DA:** Começa com desempenho muito bom, inclusive com *gap* negativo em $\rho = 10$, indicando melhoria em relação à literatura. Contudo, observa-se queda acentuada no número de soluções ótimas e aumento do *gap* a partir de $\rho = 30$. O tempo de execução apresenta pico em $\rho = 30$, diminuindo posteriormente. A maioria dos casos de INO ocorre nesse intervalo entre $\rho = 30e50$.

Em termos gerais, o GGA-CGT apresenta bom desempenho em instâncias de baixa densidade de conflitos, geralmente mais simples, e também em algumas de alta densidade,

onde a heurística utilizada para gerar a população inicial favorece soluções próximas do ótimo. No entanto, o desempenho tende a piorar em densidades intermediárias, especialmente nas classes U e TA, como evidenciado pela redução no número de soluções ótimas encontradas. Essa fragilidade também é observada na métrica INO para as classes T e DA.

Apesar das limitações observadas, o desempenho geral do GGA-CGT é satisfatório, incluindo a obtenção de uma solução melhor que a literatura em uma instância da classe DA, curiosamente a classe de pior desempenho. Com algumas melhorias pontuais, o algoritmo tem potencial para melhorar seu desempenho e resultados. Uma melhoria que se mostra promissora é a diversificação da geração da população inicial, o que poderia aumentar o espaço de busca e melhorar a eficiência do algoritmo, mesmo nas instâncias mais complexas. Essa adaptação também poderia ser incorporada à versão original do GGA-CGT, voltada ao problema clássico do BPP.

7 Conclusão

O principal objetivo deste projeto, adaptar o GGA-CGT para uma versão capaz de considerar os conflitos entre itens, foi alcançado com êxito. A implementação reafirma a versatilidade dos Algoritmos Genéticos, demonstrando seu potencial para resolver problemas combinatórios complexos. Essa adaptação também abre caminho para estudos futuros que explorem a aplicação do GGA-CGT em outras variações do BPP.

Os experimentos realizados mostraram que a geração da população inicial desempenha papel fundamental no desempenho do algoritmo. Em instâncias com baixa densidade de conflitos, a boa qualidade das soluções iniciais facilitou a convergência rápida para resultados próximos ou iguais ao ótimo. Já em cenários de maior densidade, a lógica de formação inicial da população demonstrou vantagens em casos específicos, mas também revelou limitações quando a diversidade era insuficiente para explorar adequadamente o espaço de busca. Essas evidências reforçam que a eficácia do GGA-CGT está diretamente ligada ao equilíbrio entre qualidade e diversidade da população.

Uma linha promissora para trabalhos futuros reside no desenvolvimento de uma versão adaptativa do algoritmo, capaz de ajustar automaticamente suas estratégias de acordo com a densidade de conflitos presente na instância. Essa adaptatividade permitiria otimizar tanto a formação da população inicial quanto o número de gerações sem melhora, equilibrando qualidade da solução e tempo de execução. Para instâncias com alta densidade de conflitos, estratégias mais elitistas poderiam priorizar a alocação dos itens mais conflitantes, enquanto para casos com baixa densidade, maior diversidade poderia ser explorada para ampliar a busca no espaço de soluções, algo que pode fazer sentido, inclusive, para a versão clássica do GGA-CGT.

Durante o desenvolvimento, foi possível identificar que a necessidade frequente de

validar a lista de conflitos introduz um alto custo computacional, impactando significativamente o tempo de execução. Essa limitação pode, inclusive, justificar a escassez de abordagens baseadas em Algoritmos Genéticos ou outras meta-heurísticas iterativas na literatura dedicada ao BPPC. Como linha de pesquisa futura, recomenda-se investigar alternativas para as operações de *crossover*, mutação e, principalmente, para a função de *fitness*, com o objetivo de reduzir a quantidade de validações necessárias durante a execução.

Por fim, este trabalho contribui para o avanço do estudo do Bin Packing com Conflitos ao apresentar uma adaptação consistente do GGA-CGT, evidenciando tanto seu potencial de aplicação quanto suas limitações em diferentes cenários. A análise realizada reforça a importância de compreender o papel da população inicial na eficácia de algoritmos genéticos e amplia a discussão sobre o equilíbrio entre qualidade e diversidade em meta-heurísticas. Espera-se que os resultados aqui obtidos possam fundamentar investigações futuras, servindo de referência para o desenvolvimento de métodos cada vez mais robustos e adaptativos, capazes de lidar com a complexidade inerente a problemas combinatórios de grande escala.

Referências

[1] R. C. M. A. Alto. Simulated annealing aplicado ao problema de bin packing com conflitos. Technical report, UNIVERSIDADE FEDERAL DE OURO PRETO, http://www.monografias.ufop.br/handle/35400000/7730, 2025.

[2] T. Bacci and S. Nicoloso. A heuristic algorithm for the bin packing problem with conflicts on interval graphs. Technical report, Cornell University, https://arxiv.org/abs/1707.00496, 2017.

[3] T. Bacci and S. Nicoloso. On the benchmark instances for the bin packing problem with conflicts. In *Graphs and combinatorial optimization: from theory to applications—CTW2020 proceedings*, volume 5 of *AIRO Springer Ser.*, pages 171–179. Springer, Cham, 2021. doi:10.1007/978-3-030-63072-0_14.

[4] M. Buljubašić and M. Vasquez. Consistent neighborhood search for one-dimensional bin packing and two-dimensional vector packing. *Computers & Operations Research*, 76:12–21, 2016. ISSN 0305-0548. doi:10.1016/j.cor.2016.06.009.

[5] R. Capua, Y. Frota, L. S. Ochi, and T. Vidal. A study on exponential-size neighborhoods for the bin packing problem with conflicts. *Journal of Heuristics*, 24:667–695, 2018. ISSN 1381-1231. doi:10.1007/s10732-018-9372-2.

[6] I. Doron-Arad and H. Shachnai. Approximating bin packing with conflict graphs via maximization techniques. In *Graph-theoretic concepts in computer science*, volume 14093 of *Lecture Notes in Comput. Sci.*, pages 261–275. Springer, Cham, 2023. doi:10.1007/978-3-031-43380-1_19.

[7] A. Ekici. Variable-sized bin packing problem with conflicts and item fragmentation. *Computers & Industrial Engineering*, 163:107844, 2022. ISSN 0360-8352. doi:10.1016/j.cie.2021.107844.

[8] S. Elhedhli, L. Li, M. Gzara, and J. Naoum-Sawaya. A branch-and-price algorithm for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 23(3): 404–415, 2011. ISSN 1526-5528. doi:10.1287/ijoc.1100.0406.

[9] L. Epstein and A. Levin. On bin packing with conflicts. *SIAM Journal on Optimization*, 19(3):1270–1298, 2008. ISSN 1052-6234. doi:10.1137/060666329.

[10] E. Falkenauer. The grouping genetic algorithm. In *State of the art in global optimization (Princeton, NJ, 1995)*, volume 7 of *Nonconvex Optim. Appl.*, pages 249–265. Kluwer Acad. Publ., Dordrecht, 1996. doi:10.1007/978-1-4613-3437-8_17.

[11] A. E. Fernandes Muritiba, M. Iori, E. Malaguti, and P. Toth. Algorithms for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 22(3):401–415, 2010. ISSN 1091-9856. doi:10.1287/ijoc.1090.0355.

[12] M. Gendreau and J.-Y. Potvin, editors. *Handbook of metaheuristics*, volume 272 of *International Series in Operations Research & Management Science*. Springer, Cham, 2019. ISBN 978-3-319-91085-7; 978-3-319-91086-4. doi:10.1007/978-3-319-91086-4.

[13] M. Gendreau, G. Laporte, and F. Semet. Heuristics and lower bounds for the bin packing problem with conflicts. *Computers & Operations Research*, 31(3):347–358, 2004. ISSN 0305-0548. doi:10.1016/S0305-0548(02)00195-8.

[14] J. Gonzales-San-Martin, L. Cruz-Reyes, C. Gómez-Santillán, H. Fraire, N. Rangel-Valdez, B. Dorronsoro, and M. Quiroz-Castellanos. Comparative study of heuristics for the one-dimensional bin packing problem. In *Hybrid Intelligent Systems Based on Extensions of Fuzzy Logic, Neural Networks and Metaheuristics*, volume 1096 of *Studies in Computational Intelligence*, pages 293–305. Springer, Cham, 2023. doi:10.1007/978-3-031-28999-6_19.

[15] K. Jansen. An approximation scheme for bin packing with conflicts. *Journal of Combinatorial Optimization*, 3(4):363–377, 1999. ISSN 1382-6905. doi:10.1023/A:1009871302966.

[16] A. Khanafer, F. Clautiaux, and E.-G. Talbi. New lower bounds for bin packing problems with conflicts. *European Journal of Operational Research*, 206(2):281–288, 2010. ISSN 0377-2217. doi:10.1016/j.ejor.2010.01.037.

[17] M. Maiza and M. S. Radjef. Heuristics for solving the bin-packing problem with conflicts. *Applied Mathematical Sciences*, 5(33-36):1739–1752, 2011. ISSN 1312-885X.

[18] M. Maiza, C. Guéret, P. Lemaire, and M. S. Radjef. A new lower bound for bin packing problem with general conflicts graph. In *23rd European Conference on Operational Research (EURO'XXIII)*, Bonn, Germany, 2009.

[19] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics. The Journal of Combinatorial Algorithms, Informatics and Computational Sciences*, 28(1):59–70, 1990. ISSN 0166-218X. doi:10.1016/0166-218X(90)90094-S.

[20] R. Martí, P. M. Pardalos, and M. G. C. Resende, editors. *Handbook of heuristics*. Springer, Cham, 2018. ISBN 978-3-319-07124-4; 978-3-319-07123-7; 978-3-319-07125-1. doi:10.1007/978-3-319-07124-4.

[21] M. Quiroz-Castellanos, L. Cruz-Reyes, J. Torres-Jimenez, C. Gómez S., H. J. F. Huacuja, and A. C. F. Alvim. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers & Operations Research*, 55: 52–64, 2015. ISSN 0305-0548. doi:10.1016/j.cor.2014.10.010.

[22] R. S. Sadykov and F. Vanderbeck. Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):1–12, 2013. ISSN 1526-5528. doi:10.1287/ijoc.1120.0499.

[23] V. V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, 2001. ISBN 3-540-65367-8.

[24] Y. Yuan, Y.-j. Li, and Y.-q. Wang. An improved aco algorithm for the bin packing problem with conflicts based on graph coloring model. In *International Conference on Management Science and Engineering*, pages 3–9. IEEE, 2014. doi:10.1109/ICMSE.2014.6930200.

[25] W. Zahrouni and H. Kamoun. Exact and approximate size reduction for the bin packing problem with conflicts. *Intelligent Decision Technologies*, 19(3):1822–1836, 2025. ISSN 1875-8843. doi:10.1177/18724981241307388.