BPP/Relatorio/Relatorio.md

Sumário

1. Introdução

1.1. O problema de empacotamento em caixas unidimensional (1D-BPP)

O problema de empacotamento em caixas unidimensional (1D-BPP) envolve empacotar um determinado conjunto de itens com pesos diferentes em um número mínimo de caixas, cada uma com uma capacidade fixa. O problema pode ser visto de duas maneiras:

- **Minimizar o número de caixas:** Dado um conjunto de itens e uma capacidade de caixa, encontre o menor número de caixas necessárias para empacotar todos os itens sem exceder a capacidade de qualquer caixa.
- **Minimizar a capacidade dado um número fixo de caixas (Problema Dual de Empacotamento em Caixas):** Dado um conjunto de itens e um número fixo de caixas, encontre a capacidade mínima que cada caixa deve ter para acomodar todos os itens. Essa variação também é conhecida como Problema de Escalonamento de Multiprocessadores.

O problema do empacotamento em caixas (BPP) é um problema clássico de otimização combinatória, classificado como NP-difícil, com muitas aplicações práticas em áreas como:

- Transporte e Logística
- Corte de estoque e Design de embalagens

- Alocação de recursos
- Escalonamento de máquinas

1.2. Complexidade Computacional do Problema de Empacotamento em Caixas

O problema do empacotamento em caixas (BPP) é classificado como **NP-difícil**, o que significa que não há algoritmo conhecido que possa resolvê-lo em tempo polinomial para todas as instâncias do problema.

Aqui o porquê:

- **Verificação vs. Encontrar uma Solução:** Embora seja relativamente fácil verificar se uma determinada solução para o problema de empacotamento em caixas é viável (ou seja, se todos os itens cabem nas caixas sem exceder sua capacidade), encontrar a solução ideal, que usa o menor número possível de caixas, torna-se cada vez mais difícil à medida que o número de itens e a capacidade das caixas aumentam.
- **Crescimento Exponencial de Possibilidades:** O número de maneiras possíveis de empacotar itens em caixas cresce exponencialmente com o número de itens. À medida que o tamanho do problema aumenta, mesmo os computadores mais poderosos exigiriam uma quantidade impraticável de tempo para explorar todas as combinações possíveis e garantir a descoberta da solução ideal.
- **NP-Completude:** A versão de decisão do problema de empacotamento em caixas (determinar se um conjunto de itens pode caber em um determinado número de caixas) é conhecida por ser **NP-completa**. Uma vez que a versão de otimização do problema (encontrar o número mínimo de caixas) é pelo menos tão difícil quanto a versão de decisão, ela é classificada como NP-difícil.
- **Redução de Outros Problemas NP-Completos:** A dureza NP do BPP pode ser comprovada por meio de redução, o que significa que outros problemas NP-completos conhecidos, como o problema da partição, podem ser transformados em instâncias do problema de empacotamento em caixas. Se houvesse um algoritmo de tempo polinomial para resolver o BPP, ele também poderia ser usado para resolver esses outros problemas NP-completos em tempo polinomial, o que se acredita ser impossível.

1.3. Formulação Matemática do Problema de Empacotamento em Caixas Unidimensional

1.3.1. Conjuntos

- $N = {1, ..., n}$: Conjunto de itens a serem embalados.
- $M = {1, ..., m}$: Conjunto de caixas disponíveis.

1.3.2. Parâmetros

- w_i: Peso ou tamanho do item i, onde $i ∈ N$.
- C: Capacidade de cada caixa.

1.3.3. Variáveis de Decisão

- $x_{ij} = {1 \text{se o item } j \text{é atribuído à caixa} i, 0 \text{ caso contrário}}, \text{onde } i ∈ M \text{e} j ∈ N.$
- $y_i = {1 \text{se a caixa } i \text{ é usada, } 0 \text{ caso contrário}}, \text{ onde } i ∈ M.$

1.3.4. Função Objetivo

Minimizar o número total de caixas usadas:

$$\sum_{i \in M} y_i$$

Sujeito às seguintes restrições:

- Restrição de Capacidade: O peso total dos itens atribuídos a uma caixa não pode exceder sua capacidade:

$$\sum_{j \in N} w_j x_{ij} \leq C y_i, \text{ para todo } i \in M$$

- **Restrição de Atribuição:** Cada item deve ser atribuído a exatamente uma caixa:

$$\sum_{i \in M} x_{ij} = 1, \text{ para todo } j \in N$$

- **Restrições Binárias:** As variáveis de decisão são binárias:

$$x_{ij} \in \{0, 1\}, \text{ para todo } i \in M, j \in N$$

$$y_i \in \{0, 1\}, \text{ para todo } i \in M$$

Explicação:

Esta formulação visa encontrar o número mínimo de caixas (m) necessárias para embalar todos os itens, sujeito às restrições. A função objetivo minimiza a soma das variáveis y_i, que representam se uma caixa é usada ou não. As restrições garantem que a capacidade de cada caixa seja respeitada e que cada item seja atribuído a exatamente uma caixa.

Conceitos Chave:

- **NP-difícil:** O empacotamento em caixas é NP-difícil, o que significa que nenhum algoritmo conhecido pode resolvê-lo em tempo polinomial para todas as instâncias. Portanto, algoritmos aproximados (heurísticas) são frequentemente usados para encontrar soluções quase ideais em um tempo razoável.
- **Variáveis de Decisão:** Essas variáveis, x_{ij} e y_i, representam as decisões que estão sendo tomadas no problema e assumem valores binários (0 ou 1) para indicar a ação escolhida.
- **Função Objetivo:** Essa função expressa matematicamente o objetivo do problema de otimização, que é minimizar o número de caixas usadas.
- **Restrições:** Essas desigualdades ou igualdades definem as soluções viáveis para o problema, garantindo que cada solução satisfaça os requisitos do problema.

Esta formulação matemática fornece uma representação concisa do 1D-BPP, permitindo a aplicação de técnicas de otimização para encontrar soluções.

1.4. Abordagens de Solução

- **Algoritmos heurísticos e metaheurísticos:** Algoritmos heurísticos e metaheurísticos são frequentemente empregados para instâncias grandes do BPP para encontrar boas soluções de forma eficiente. Algumas das heurísticas comumente usadas incluem First-Fit Decreasing (FFD) e Best-Fit Decreasing (BFD). Esses algoritmos fornecem soluções aproximadas, mas podem nem sempre garantir o valor ótimo.

1.4.1. Técnicas de Melhoria

Pesquisadores exploraram diversas técnicas para aprimorar o desempenho das soluções BPP, incluindo:

- **Critério de Dominância:** Esta técnica identifica e fixa caixas que dominam outras com base em seus itens embalados, reduzindo significativamente o tamanho do problema e otimizando o processo de resolução.
- **Procedimentos de Redução:** Estes procedimentos avançados simplificam a instância do problema ao identificar e eliminar itens ou caixas redundantes, seguindo regras específicas e complexas, o que resulta em uma representação mais enxuta e eficiente do problema original.
- **Abordagens Híbridas:** A combinação sinérgica de diferentes heurísticas e metaheurísticas — como algoritmos genéticos, busca tabu e simulated annealing — tem demonstrado resultados excepcionalmente promissores, aproveitando as forças de cada método para superar suas limitações individuais.
- **Processamento Paralelo:** A paralelização de algoritmos evolutivos, como os Algoritmos Genéticos de Agrupamento (GGA) pode melhorar substancialmente a exploração do espaço de busca e a qualidade da solução, permitindo uma busca mais abrangente e eficiente em menos tempo.

1.4.2. O que é Meta-Heuristica?

Meta-heurísticas são técnicas avançadas de otimização que visam encontrar soluções aproximadas da ótima para problemas extremamente complexos, nos quais a obtenção da solução exata seria inviável devido ao tempo computacional proibitivo. Esses métodos são particularmente relevantes em situações onde a natureza combinatória do problema impede a aplicação de algoritmos exatos, requerendo abordagens que equilibrem a qualidade da solução com o tempo de execução.

Diferentemente dos algoritmos tradicionais que realizam uma busca exaustiva por todas as soluções possíveis, as meta-heurísticas utilizam estratégias inteligentes para guiar a exploração do espaço de soluções. Essas estratégias fazem uso de mecanismos sofisticados de exploração e de intensificação, o que permite escapar de mínimos locais e continuar a busca por soluções melhores. Esta característica as diferencia das heurísticas tradicionais, que frequentemente ficam presas em ótimos locais devido à sua limitada capacidade de exploração.

A flexibilidade das meta-heurísticas também é um ponto crucial, pois permite que elas sejam aplicadas a uma ampla gama de problemas, independentemente da estrutura particular de cada problema. Isso contribui para sua ampla adoção em diversos campos, tanto na pesquisa acadêmica quanto na indústria. Ao integrar estratégias como memória adaptativa, diversificação e intensificação da busca, as meta-heurísticas conseguem se adaptar dinamicamente ao comportamento do problema ao longo do processo de otimização, proporcionando soluções de alta qualidade em um tempo computacional razoável.

2. Algoritmo Genético de Agrupamento (GGA) e hibridização

2.1. Motivação para o GGA

Durante a revisão de artigos acadêmicos, o Algoritmo Genético de Agrupamento (GGA) destacou-se por sua eficácia na resolução do problema do empacotamento em caixas (BPP). O GGA superou outros algoritmos em termos de qualidade das soluções, tempo de execução e eficiência no uso dos recursos computacionais.

Os algoritmos genéticos tradicionais (GAs) enfrentam dificuldades em problemas de agrupamento, como o problema do empacotamento em caixas (BPP), devido a limitações na representação dos dados e à natureza dos operadores genéticos padrão. O GGA supera esses desafios por meio de:

- **Codificação Baseada em Grupos**: O GGA utiliza uma codificação em que os genes representam grupos de itens atribuídos à mesma caixa, refletindo diretamente a estrutura da solução do BPP.
- **Operadores Genéticos Especializados**: O GGA emprega operadores de cruzamento e mutação que manipulam grupos de itens, promovendo uma exploração mais eficaz do espaço de soluções.

2.2. Descrição do Algoritmo

O Algoritmo Genético de Agrupamento (GGA), proposto por Falkenauer Emanuel (1993), é uma abordagem evolutiva desenvolvida especificamente para resolver problemas de agrupamento de dados, sendo mais eficaz que os algoritmos genéticos tradicionais. O GGA utiliza operadores avançados, como o cruzamento baseado em grupos e a mutação adaptativa, que são projetados para preservar as características de alta qualidade das soluções durante o processo evolutivo.

Além disso, o GGA emprega técnicas de reprodução controlada, como a reprodução seletiva balanceada, que mantêm um equilíbrio entre pressão seletiva e diversidade populacional, permitindo uma exploração e uma exploração equilibradas do espaço de busca. Diferente dos algoritmos genéticos tradicionais, o GGA adota operadores especializados que lidam diretamente com a natureza dos agrupamentos, tornando-o mais adequado para problemas de alocação e particionamento de conjuntos de dados.

Essas características tornam o GGA particularmente eficiente em obter soluções de alta qualidade em um curto período de tempo.

2.2.1. Principais Características do GGA

- **Representação Cromossômica**:
  - **Seção de Elementos**: Representa os itens a serem embalados.
  - **Seção de Agrupamento**: Representa as caixas e os grupos de itens atribuídos a cada caixa, com comprimento variável.
- **Geração da População Inicial**:
  - Heurísticas como First-Fit (FF) ou Best-Fit Decreasing (BFD) são utilizadas para criar uma população inicial de alta qualidade. Evidências de diversos estudos, como Quiroz-Castellanos et al. (2015), Kucukyilmaz e Kiziloz (2018), Layeb e Chenche (2012), e Luo et al. (2017), mostram que essas heurísticas garantem uma distribuição eficiente dos itens desde o início, contribuindo significativamente para a eficácia dos algoritmos de empacotamento.
- **Função de Aptidão (Fitness)**:
  - Avalia a qualidade de cada solução, considerando a ocupação e a eficiência das caixas.
- **Seleção, Cruzamento e Mutação**:
  - **Seleção**: Favorece cromossomos com altos valores de aptidão.
  - **Cruzamento**: Recombina cromossomos pais, preservando e propagando agrupamentos eficazes de itens.
  - **Mutação**: Introduz pequenas alterações aleatórias para manter a diversidade.
- **Otimização Local**:
  - Heurísticas de busca local, como Tabu Search, combinadas com First-Fit Decreasing (FFD) ou Best-Fit Decreasing (BFD), podem ser incorporadas para refinar as soluções e melhorar a qualidade final. A busca local complementa a busca global do GGA, explorando detalhadamente regiões promissoras do espaço de soluções. Por exemplo, a busca local pode envolver a troca de itens entre caixas para melhorar a ocupação, resultando em um refinamento incremental.

2.2.2. Vantagens do GGA

- **Eficácia em Problemas de Agrupamento**: A codificação e os operadores especializados tornam o GGA particularmente adequado para o BPP e outros problemas de agrupamento.
- **Qualidade de Solução Melhorada**: A combinação de busca global e otimização local permite alcançar soluções de alta qualidade.

2.2.3. Variações e Extensões

- **GGAs Híbridos**: Combinação com outras metaheurísticas.
- **GGAs Paralelos**: Distribuição da carga de trabalho para reduzir o tempo de execução.

2.2.4. Pseudo-Algoritmo do Grouping Genetic Algorithm

```
Início

    // Parâmetros do Algoritmo Genético

    População_Tamanho ← N

    Geração_Max ← M

    Taxa_Cruzamento ← C%

    Taxa_Mutação ← μ%

    Melhor_Solução ← NULL

    // Passo 1: Inicialização

    População ← Gerar_População_Inicial(População_Tamanho)

    // Avaliar a aptidão inicial

    Para cada Indivíduo em População faça
```

```
    Indivíduo.Aptidão ← Avaliar_Aptidão(Indivíduo)
FimPara
// Passo 2: Evolução através das gerações
Para Geração de 1 até Geração_Max faça
    Nova_População ← []
    Enquanto tamanho(Nova_População) < População_Tamanho faça
        // Seleção dos pais
        Pai1 ← Selecionar_Pai(População)
        Pai2 ← Selecionar_Pai(População)
        // Cruzamento
        Se Random() < Taxa_Cruzamento então
            [Filho1, Filho2] ← Cruzar(Pai1, Pai2)
        Senão
            [Filho1, Filho2] ← [Pai1, Pai2]
        FimSe
        // Mutação
        Se Random() < Taxa_Mutação então
            Filho1 ← Mutar(Filho1)
        FimSe
        Se Random() < Taxa_Mutação então
            Filho2 ← Mutar(Filho2)
        FimSe
        // Aplicar busca local para refinar os filhos
        Filho1 ← Busca_Local(Filho1)
        Filho2 ← Busca_Local(Filho2)
        // Adicionar filhos à nova população
        Adicionar(Nova_População, Filho1)
        Adicionar(Nova_População, Filho2)
    FimEnquanto
    // Passo 3: Avaliação da nova população
    Para cada Indivíduo em Nova_População faça
        Indivíduo.Aptidão ← Avaliar_Aptidão(Indivíduo)
    FimPara
    // Passo 4: Seleção da população para a próxima geração
    População ← Selecionar_Sobreviventes(População, Nova_População)
    // Atualizar a melhor solução encontrada
```

```
        Melhor_Solução_Geração ← Encontrar_Melhor(População)
        Se Melhor_Solução for NULL ou Melhor_Solução_Geração.Aptidão > Melhor_Solução.Aptidão então
            Melhor_Solução ← Melhor_Solução_Geração
        FimSe
        // Critério de parada antecipada se convergir
        Se Convergência(População) então
            Interromper
        FimSe
    FimPara
    // Resultado final
    Retornar Melhor_Solução
Fim
```

2.3. Hibridização do GGA com Outras Heurísticas

A hibridização dos Algoritmos Genéticos de Agrupamento (GGAs) com outras heurísticas é uma abordagem promissora para resolver o problema do empacotamento em caixas (BPP) e outros problemas de agrupamento. Diversos artigos indicam que esses métodos híbridos são altamente eficazes, conforme discutido em *Alvim et al.* (2004), *Scholl et al.* (1997), e *Quiroz-Castellanos et al.* (2015), que destacam a eficácia dos métodos HI_BP, BISON e GGA-CGT, respectivamente.

2.3.1. Vantagens do GGA como Base

- **Representação Baseada em Grupos**: Os GGAs se destacam na representação de problemas de agrupamento porque sua estrutura cromossômica corresponde diretamente à estrutura do problema. Cada gene em um cromossomo GGA representa um grupo de itens (uma caixa no caso do BPP), capturando naturalmente o relacionamento entre os itens dentro de um grupo. Isso difere dos GAs padrão, nos quais a representação não corresponde diretamente ao aspecto de agrupamento do problema, resultando na interrupção de bons agrupamentos durante o cruzamento e a mutação.
- **Operadores Especializados para Agrupamento**: Os GGAs empregam operadores de cruzamento e mutação projetados especificamente para trabalhar com grupos de itens em vez de itens individuais. Essa característica é crucial para preservar e explorar agrupamentos promissores durante o processo evolutivo. No contexto do BPP, o cruzamento pode envolver a troca de caixas inteiras entre soluções parentais, enquanto a mutação pode envolver a remoção e reinserção de grupos de itens, utilizando heurísticas para reembalá-los.

2.3.2. Benefícios da Hibridização com Outras Heurísticas

- **Busca Local Aprimorada**: Embora os GGAs se destaquem na busca global, eles podem ser ainda mais eficazes ao integrar heurísticas de busca local para otimizar soluções dentro de cada caixa ou refinar o arranjo das caixas.
- **Qualidade da População Inicial Melhorada**: O uso de heurísticas para gerar a população inicial para o GGA pode melhorar significativamente o desempenho. Heurísticas de construção, como First-Fit, podem ser usadas como decodificadores para garantir a viabilidade das soluções desde o início.
- **Abordagem de Desafios Específicos**: A hibridização permite adaptar os GGAs para lidar com desafios específicos no BPP e em outros problemas de agrupamento:
  - A transmissão dos melhores genes é realizada por meio de novos operadores de agrupamento, enquanto uma nova técnica de reprodução controla a exploração do espaço de busca e previne a convergência prematura do algoritmo.
  - Introdução de uma estratégia para limitar o tamanho do problema, reduzindo a redundância no espaço de busca.

2.3.3. Exemplos de GGAs Híbridos Bem-sucedidos

As implementações bem-sucedidas de GGA híbrido incluem:

- **GGA com Transmissão de Genes Controlada (GGA-CGT)**:
  - Uma heurística de empacotamento inteligente (FF-ñ).
  - Operadores de agrupamento inteligentes para promover a transmissão de bons genes.
  - Procedimentos de rearranjo para melhorar as soluções.
  - Uma técnica de reprodução controlada para equilibrar exploração e exploração.
- **Algoritmo Genético Baseado em Heurística com Bloqueio Dinâmico (HBGAwDB) (***Cardoso Silva, Aluísio and Carlos Cristiano Hasenclever Borges, 2019*)
- **Algoritmo Genético de Agrupamento Paralelo de Ilha (IPGGA)** (*Kucukyilmaz and Kiziloz, 2018)*

2.3.4. Considerações dos Parâmetros

- **Ajuste de Parâmetros**: Para o ajuste dos parâmetros da GGA, foi-se usada otimização Bayesiana, utilizando a biblioteca `Optuna` . Utilizando algumas instâncias do problema para achar valores interessantes para os parâmetros iniciais da GGA.

  Também colocamos como opcional, ao passar um dicionário com os parâmetros, assim, podendo ao usuário poder ajustar como quiser.

2.4. Tabu Search

Para a hibridização da GGA, foi escolhida a meta-heurística Tabu Search, com o objetivo de realizar uma busca extensiva a partir de uma solução encontrada pela GGA.

Tabu Search (TS) é uma meta-heurística sofisticada desenvolvida para resolver problemas complexos de otimização, especialmente no contexto de problemas combinatórios. Concebida por Fred Glover na década de 1980, a Tabu Search busca melhorar soluções iniciais de forma iterativa, empregando estratégias que previnem a estagnação em ótimos locais — uma limitação comum em métodos tradicionais de busca local.

A TS diferencia-se como um método de otimização pelo uso de estruturas de memória que orientam a exploração do espaço de soluções, permitindo a obtenção de soluções de alta qualidade em contextos altamente complexos. Frequentemente, a TS é combinada com outras heurísticas, potencializando suas operações ao direcionar a exploração para áreas promissoras do espaço de busca. A capacidade de memória da TS permite uma exploração que vai além da avaliação imediata da função objetivo e das soluções vizinhas, sendo particularmente eficaz em problemas de otimização combinatória em larga escala.

2.4.1. Elementos-chave da Tabu Search

1. **Neighborhood**: O conceito de vizinhança é fundamental na Tabu Search, consistindo no conjunto de soluções que podem ser obtidas a partir de modificações na solução atual. A busca na vizinhança visa identificar uma solução que represente uma melhoria em relação ao estado atual.
2. **Moves**: Movimentos são as transições realizadas entre soluções dentro da vizinhança. Cada movimento é caracterizado por atributos que definem seu impacto na qualidade da solução, permitindo um controle mais refinado sobre o processo de busca.
3. **Tabu List**: A lista tabu é uma estrutura de memória responsável por registrar características de soluções ou movimentos recentemente visitados, de modo a evitar ciclos e promover a exploração de novas regiões do espaço de busca. Ao proibir movimentos que retornariam a estados recentemente visitados, a lista tabu garante diversificação na busca.
4. **Tabu Tenure**: A tenure define a duração pela qual um atributo de movimento ou uma solução permanece na lista tabu. Uma tenure mais longa favorece a diversificação, enquanto uma tenure curta promove a intensificação da busca. A tenure pode ser ajustada dinamicamente, dependendo das características do problema e do progresso da busca.
5. **Aspiration Criteria**: Critérios de aspiração são regras que permitem a aceitação de movimentos classificados como tabu, desde que atendam a determinadas condições, como proporcionar uma solução melhor do que a melhor solução já encontrada. Isso flexibiliza a busca e evita que restrições tabu sejam excessivamente limitantes.
6. **Intensification**: A estratégia de intensificação concentra a busca em regiões promissoras do espaço de solução, explorando-as de maneira mais aprofundada para identificar soluções de alta qualidade. A intensificação é crucial quando se deseja refinar uma solução promissora.
7. **Diversification**: A estratégia de diversificação visa explorar novas regiões do espaço de solução, evitando a convergência prematura para ótimos locais. Ao incentivar movimentos que levam a áreas pouco exploradas, a diversificação aumenta a robustez do algoritmo frente à complexidade do espaço de busca.

2.4.2. Pseudo-Algoritmo de Tabu Search

```
Início

    // Parâmetros do algoritmo Tabu Search
    Solução_Inicial ← Solução fornecida pelo GGA
    Melhor_Solução ← Solução_Inicial
    Melhor_Aptidão ← Avaliar_Aptidão(Melhor_Solução)

    Iteração ← 0
    Tabu_List ← Fila de tamanho máximo (Tabu_Tenure)
    Tabu_Set ← Conjunto vazio

    // Enquanto o número de iterações for menor que Max_Iterações
    Enquanto Iteração < Max_Iterações faça

        Vizinho_Encontrado ← Falso

        // Gerar vizinhança para a solução atual
        Vizinhos ← Gerar_Vizinhança(Solução_Atual)

        Para cada (Vizinho, Movimento, Aptidão) em Vizinhos faça

            // Se o movimento não está na Tabu_Set ou é melhor que a melhor aptidão
            Se Movimento não está em Tabu_Set ou Aptidão < Melhor_Aptidão então

                // Atualizar Tabu_List e Tabu_Set
                Adicionar Tabu_List(Movimento)
```

```
                Adicionar Tabu_Set(Movimento)

                Se o tamanho de Tabu_List > Tabu_Tenure então
                    Remover Movimento_Mais_Antigo de Tabu_List e Tabu_Set
                FimSe

                // Atualizar solução atual
                Solução_Atual ← Vizinho

                // Atualizar a melhor solução se o vizinho for melhor
                Se Aptidão < Melhor_Aptidão então
                    Melhor_Solução ← Vizinho
                    Melhor_Aptidão ← Aptidão
                FimSe

                Vizinho_Encontrado ← Verdadeiro
                Parar // Move para a próxima iteração

            FimSe

        FimPara

        // Se nenhum vizinho aceitável for encontrado, parar a busca
        Se Vizinho_Encontrado = Falso então
            Parar
        FimSe

        Iteração ← Iteração + 1

    FimEnquanto

    // Retorna a melhor solução encontrada
    Retornar Melhor_Solução
    Fim

    // Função para gerar a vizinhança da solução atual
    Função Gerar_Vizinhança(Solução_Atual)

    Vizinhos ← Lista vazia
    Tamanho_Solução ← Tamanho da solução (número de contêineres)

    Tentativas ← 0
    Max_Tentativas ← Max_Vizinhos * 10 // Limite para evitar loops

    Enquanto tamanho(Vizinhos) < Max_Vizinhos e Tentativas < Max_Tentativas faça
        Tentativas ← Tentativas + 1

        // Escolher dois contêineres aleatórios
        (i, j) ← Escolher aleatoriamente dois índices

        Contêiner_i ← Solução_Atual[i]
        Contêiner_j ← Solução_Atual[j]

        // Se o contêiner_i estiver vazio, continue
        Se Contêiner_i estiver vazio então
            Continue
        FimSe

        // Escolher um item aleatório do contêiner_i
        Item ← Escolher item aleatório de Contêiner_i

        // Se o contêiner_j tiver espaço suficiente para o item
        Se Contêiner_j tiver espaço suficiente para o Item então

            // Criar nova solução aplicando o movimento
            Nova_Solução ← Cópia de Solução_Atual
            Nova_Solução[i] ← Cópia de Contêiner_i
            Nova_Solução[j] ← Cópia de Contêiner_j

            // Remover o item do contêiner_i e adicionar ao contêiner_j
            Nova_Solução[i].Remover_Item(Item)
```

```
            Nova_Solução[j].Adicionar_Item(Item)

            // Remover contêineres vazios
            Nova_Solução ← Remover_Contêineres_Vazios(Nova_Solução)

            // Avaliar a nova solução
            Movimento ← (Item, i, j)
            Aptidão ← Avaliar_Aptidão(Nova_Solução)

            // Adicionar a nova solução e seu movimento à lista de vizinhos
            Adicionar (Nova_Solução, Movimento, Aptidão) à Vizinhos

        FimSe

    FimEnquanto

    Retornar Vizinhos

FimFunção
```

3. Solução

3.1. Metaheurísticas Utilizadas

Neste trabalho, adotamos diversas heurísticas para otimizar a busca por soluções. A heurística Best-Fit Decreasing (BFD) foi empregada para gerar uma solução inicial de forma rápida, distribuindo itens de maneira sequencial.

O algoritmo `Tabu Search` foi então aplicado para refinar essas soluções, evitando a estagnação em ótimos locais por meio da manutenção de uma lista de soluções proibidas (tabus). Essas técnicas, em conjunto com operadores genéticos, ajudaram a explorar de forma eficiente o espaço de busca.

A escolha dessas metaheurísticas se deve a alguns motivos.

Como precisávamos de uma solução inicial, a BFD se destacou em relação à First-Fit Decreasing (FFD)(que foi usada num primeiro momento), por ordenar os itens em ordem decrescente antes de alocá-los. Isso permite que os itens maiores sejam colocados primeiro, reduzindo o espaço residual, assim, maximizando a utilização de cada contêiner.

A FFD, por outro lado, coloca cada item no primeiro contêiner disponível que possa acomodá-lo, sem considerar a otimização do espaço restante.

A BFD também se destaca por adaptar-se melhor a cenários com itens de tamanhos variados. Esse conjunto de fatores levou à escolha pela BFD.

Quanto ao Tabu Search, ele se destaca na exploração e refinamento de ótimos locais, além de proporcionar oportunidades para escapar de ótimos locais subótimos e realizar uma busca global.

3.1.1. Pseudo-Algoritmo da Best-Fit Decreasing (BFD)

```
Início
    // Passo 1: Ordenar os itens em ordem decrescente de tamanho
    Ordenar_Itens_Decrescente(Itens)

    // Passo 2: Inicializar lista de contêineres vazios
    Contêineres ← Lista vazia

    // Passo 3: Para cada item, tentar colocá-lo no melhor contêiner disponível
    Para cada Item em Itens faça

        Melhor_Contêiner ← NULL
        Melhor_Espaço_Restante ← Infinito

        // Passo 4: Verificar cada contêiner existente
        Para cada Contêiner em Contêineres faça

            Espaço_Restante ← Contêiner.Espaço_Restante()

            // Passo 5: Se o contêiner pode acomodar o item
            Se Espaço_Restante >= Tamanho do Item então

                // Verificar se este contêiner é o melhor (menor espaço restante)
```

```
                Se Espaço_Restante < Melhor_Espaço_Restante então
                    Melhor_Contêiner ← Contêiner
                    Melhor_Espaço_Restante ← Espaço_Restante
                FimSe

            FimSe

        FimPara

        // Passo 6: Se foi encontrado um contêiner adequado, adicionar o item
        Se Melhor_Contêiner ≠ NULL então
            Melhor_Contêiner.Adicionar_Elemento(Item)

        // Caso contrário, criar um novo contêiner e adicionar o item
        Senão
            Novo_Contêiner ← Criar novo Contêiner com capacidade máxima
            Novo_Contêiner.Adicionar_Elemento(Item)
            Adicionar Novo_Contêiner à lista de Contêineres

        FimSe

    FimPara

    // Passo 7: Retornar a lista de contêineres utilizados
    Retornar Contêineres
Fim
```

3.2. Implementação Inicial

A implementação inicial do GGA utilizou operadores clássicos do algoritmo, como a seleção por torneio, cruzamento de ponto único e mutação por troca (swap mutation).

Para gerar uma solução inicial, aplicamos a heurística Best-Fit Decreasing (BFD), que proporcionou soluções promissoras. Essa abordagem simples nos permitiu estabelecer uma base sólida para experimentação subsequente.

Para fins de comparação, utilizamos a biblioteca `OR-Tools` da Google nas mesmas instâncias de teste. (Documentação oficial da OR-Tools (Google Optimization Tools)).

```
Generation 194: Best Fitness = 45
Generation 195: Best Fitness = 45
Generation 196: Best Fitness = 45
Generation 197: Best Fitness = 45
Generation 198: Best Fitness = 45
Generation 199: Best Fitness = 45
Best solution: [[60, 60], [59, 59], [59, 59], [58, 58], [58, 58], [58, 58], [57, 58], [57, 57], [57, 57], [57, 57], [57, 57], [56, 56]
[55, 55, 40], [41, 55, 54], [42, 54, 54], [54, 53, 43], [44, 53, 53], [53, 53, 44], [53, 44, 53], [44, 53, 53], [53, 43, 53], [53, 52,
2, 52], [46, 52, 52], [46, 52, 52], [52, 52, 46], [52, 46, 52], [48, 51, 51], [49, 50, 51], [50, 50, 50], [50, 50, 50], [49, 49, 49],
, [49, 49, 48], [48, 48, 48], [48, 48, 47], [47, 47, 47], [47, 45, 45], [45, 45, 45], [43, 43, 43], [43, 42, 42], [41, 41, 41], [40, 4
]
solution size:  45
```

Para a primeira entrada, obtivemos 45 contêineres utilizados, em comparação com os 41 contêineres obtidos pelo algoritmo do `OR-Tools`. Isso demonstrou a eficácia inicial do GGA, embora tenha evidenciado áreas onde há potencial para melhorias adicionais.

A fim de aprimorar os resultados, incorporamos o algoritmo `Tabu Search`, que foi empregado para refinar as soluções parciais geradas pelo GGA. Cada indivíduo gerado passava por um processo de melhoria utilizando `Tabu Search`, permitindo uma busca local mais intensiva e direcionada em regiões promissoras do espaço de soluções.

Em seguida, exploramos diferentes combinações de operadores para melhorar a performance do algoritmo. Introduzimos métodos de seleção como o Roulette Wheel Selection e o Stoic Tournament Selection, assim como o operador de cruzamento Multi-Point Crossover. Também realizamos paralelização na CPU para acelerar o processo de busca. Essas modificações resultaram em soluções de maior qualidade, embora com um custo computacional ligeiramente superior, como vemos a baixo.

```
Melhor solução encontrada na Grouping Genetic Algorithm:
================================================================================================
Contêiner 1: Container([57, 50, 43], usado: 150/150)
Contêiner 2: Container([46, 58, 46], usado: 150/150)
Contêiner 3: Container([48, 53, 49], usado: 150/150)
Contêiner 4: Container([50, 52, 40], usado: 142/150)
Contêiner 5: Container([56, 54, 40], usado: 150/150)
Contêiner 6: Container([52, 54, 41], usado: 147/150)
Contêiner 7: Container([53, 48, 45], usado: 146/150)
Contêiner 8: Container([56, 57], usado: 113/150)
Contêiner 9: Container([55, 48, 44], usado: 147/150)
Contêiner 10: Container([55, 51, 42], usado: 148/150)
Contêiner 11: Container([57, 57], usado: 114/150)
Contêiner 12: Container([52, 50, 47], usado: 149/150)
Contêiner 13: Container([53, 53, 40], usado: 146/150)
Contêiner 14: Container([53, 53, 40], usado: 146/150)
Contêiner 15: Container([49, 45, 49], usado: 143/150)
Contêiner 16: Container([51, 48, 51], usado: 150/150)
Contêiner 17: Container([58, 57], usado: 115/150)
Contêiner 18: Container([52, 56, 41], usado: 149/150)
Contêiner 19: Container([53, 53, 40], usado: 146/150)
Contêiner 20: Container([48, 60, 41], usado: 149/150)
Contêiner 21: Container([57, 58], usado: 115/150)
Contêiner 22: Container([58, 53], usado: 111/150)
Contêiner 23: Container([52, 48, 50], usado: 150/150)
Contêiner 24: Container([59, 46, 44], usado: 149/150)
Contêiner 25: Container([59, 46, 43], usado: 148/150)
Contêiner 26: Container([49, 50, 44], usado: 143/150)
Contêiner 27: Container([55, 52, 43], usado: 150/150)
Contêiner 28: Container([53, 53, 41], usado: 147/150)
Contêiner 29: Container([51, 54, 45], usado: 150/150)
Contêiner 30: Container([59, 58], usado: 117/150)
Contêiner 31: Container([49, 50, 45], usado: 144/150)
Contêiner 32: Container([45, 53, 52], usado: 150/150)
Contêiner 33: Container([42, 48, 58], usado: 148/150)
Contêiner 34: Container([40, 60, 49], usado: 149/150)
Contêiner 35: Container([52, 45, 53], usado: 150/150)
Contêiner 36: Container([53, 50, 47], usado: 150/150)
Contêiner 37: Container([52, 55, 43], usado: 150/150)
Contêiner 38: Container([47, 52, 49], usado: 148/150)
Contêiner 39: Container([46, 48, 53], usado: 147/150)
Contêiner 40: Container([57, 45, 47], usado: 149/150)
Contêiner 41: Container([51, 49, 49], usado: 149/150)
Contêiner 42: Container([58, 45, 46], usado: 149/150)
================================================================================================
Quantidade de Contêiners usados:  42
================================================================================================
Tempo de solução: 34.25 segundos
```

Posteriormente, implementamos a mutação de inversão (Inversion Mutation) e utilizamos técnicas de otimização de hiperparâmetros para identificar as configurações mais adequadas para os operadores do algoritmo. Isso nos permitiu obter resultados quase ótimos e, em alguns casos, atingir o valor ótimo para instâncias específicas.

```
Contêiner 5: Container([56, 54, 40], usado: 150/150)
Contêiner 6: Container([52, 54, 41], usado: 147/150)
Contêiner 7: Container([53, 48, 45], usado: 146/150)
Contêiner 8: Container([56, 57], usado: 113/150)
Contêiner 9: Container([55, 48, 44], usado: 147/150)
Contêiner 10: Container([55, 51, 42], usado: 148/150)
Contêiner 11: Container([57, 57], usado: 114/150)
Contêiner 12: Container([52, 50, 47], usado: 149/150)
Contêiner 13: Container([53, 53, 41], usado: 147/150)
Contêiner 14: Container([53, 49, 48], usado: 150/150)
Contêiner 15: Container([49, 58, 42], usado: 149/150)
Contêiner 16: Container([56, 52, 40], usado: 148/150)
Contêiner 17: Container([53, 57, 40], usado: 150/150)
Contêiner 18: Container([59, 51, 40], usado: 150/150)
Contêiner 19: Container([52, 45, 53], usado: 150/150)
Contêiner 20: Container([48, 58, 40], usado: 146/150)
Contêiner 21: Container([53, 52, 43], usado: 148/150)
Contêiner 22: Container([53, 52, 41], usado: 146/150)
Contêiner 23: Container([48, 50, 46], usado: 144/150)
Contêiner 24: Container([54, 46, 50], usado: 150/150)
Contêiner 25: Container([59, 59], usado: 118/150)
Contêiner 26: Container([55, 52, 43], usado: 150/150)
Contêiner 27: Container([53, 54, 43], usado: 150/150)
Contêiner 28: Container([47, 59, 43], usado: 149/150)
Contêiner 29: Container([58, 49, 41], usado: 148/150)
Contêiner 30: Container([50, 45, 53], usado: 148/150)
Contêiner 31: Container([50, 45, 53], usado: 148/150)
Contêiner 32: Container([53, 45, 52], usado: 150/150)
Contêiner 33: Container([47, 60, 43], usado: 150/150)
Contêiner 34: Container([52, 53, 43], usado: 148/150)
Contêiner 35: Container([53, 52, 45], usado: 150/150)
Contêiner 36: Container([55, 47, 46], usado: 148/150)
Contêiner 37: Container([52, 49, 49], usado: 150/150)
Contêiner 38: Container([48, 57, 45], usado: 150/150)
Contêiner 39: Container([51, 47, 51], usado: 149/150)
Contêiner 40: Container([47, 45, 57], usado: 149/150)
Contêiner 41: Container([49, 43, 58], usado: 150/150)
================================================================
Quantidade de contêineres usados:  41
================================================================
Tempo total de solução: 21.62 segundos
```

3.2.1 Aprimoramento

Após os testes iniciais, aprimoramos as soluções testando diferentes operadores, trazendo uma abordagem um pouco diferente, mas que levassem a um resultado satisfatório. Eles são apenas citados nos anexos desse documento, como uma lista, dos quais idealizamos e implementamos.

Também escolhemos por abandonar as comparações com o OR-Tools, pois, enquanto escalávamos em questão de número de itens, o OR-Tools também subia exponencialmente o tempo de execução para retornar uma resposta.

Além de otimizar o tempo de execução, alterando as implementações com estrutura de dados mais eficientes, para encontrar um resultado em um tempo consideravelmente bom.

Isso é mostrado logo abaixo nos testes e Benchmark que obtivemos com cada uma dos operadores implementados

3.3. Testes e Benchmark

Para os testes dos algoritmos implementados, juntamente dos operadores, foram usados as Instâncias criadas por *Scholl at al.* e *M. Darlone et al.* que se encontram nesse link no GitHub Heuristicas/BPP/BPPInstances at main · Marcux777/Heuristicas

3.3.1. Configuração da Maquina que foi testada a implementação

Processador: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz

Memória Ram: 8,00 GB (utilizável: 7,77 GB)

3.3.2. Benchmark

Para facilitar a visualização de cada operador, vamos mostrar os testes apenas para a instância N1W1B1R5 fornecida por *Scholl at al.*

Mutação	Cruzamento	Seleção	Tempo	Número de Contêineres
Gaussiana	**Multi-Point**	**Stoic Tournament Selection**	28.35 segundos	18
Gaussiana	**Multi-Point**	**Tournament Selection**	25.20 segundos	18
Gaussiana	**Multi-Point**	**Roulette Wheel**	28.92 segundos	17
Gaussiana	**Single-Point**	**Stoic Tournament Selection**	8.24 segundos	18
Gaussiana	**Single-Point**	**Tournament Selection**	7.66 segundos	18
Gaussiana	**Single-Point**	**Roulette Wheel**	18.08 segundos	18
Gaussiana	**PMX**	**Stoic Tournament Selection**	—	—
Gaussiana	**PMX**	**Tournament Selection**	—	—
Gaussiana	**PMX**	**Roulette Wheel**	—	—
————	————	————	————	————
Scramble	**Multi-Point**	**Stoic Tournament Selection**	12.47 segundos	18
Scramble	**Multi-Point**	**Tournament Selection**	29.86 segundos	17
Scramble	**Multi-Point**	**Roulette Wheel**	38.16 segundos	18
Scramble	**Single-Point**	**Stoic Tournament Selection**	37.02 segundos	18
Scramble	**Single-Point**	**Tournament Selection**	6.15 segundos	18
Scramble	**Single-Point**	**Roulette Wheel**	33.84 segundos	18
Scramble	**PMX**	**Stoic Tournament Selection**	—	—
Scramble	**PMX**	**Tournament Selection**	—	—
Scramble	**PMX**	**Roulette Wheel**	—	—
————	————	————	————	————
Swap	**Multi-Point**	**Stoic Tournament Selection**	17.86 segundos	18
Swap	**Multi-Point**	**Tournament Selection**	24.87 segundos	18
Swap	**Multi-Point**	**Roulette Wheel**	53.45 segundos	17
Swap	**Single-Point**	**Stoic Tournament Selection**	10.12 segundos	18
Swap	**Single-Point**	**Tournament Selection**	14.65 segundos	18
Swap	**Single-Point**	**Roulette Wheel**	8.96 segundos	18
Swap	**PMX**	**Stoic Tournament Selection**	—	—
Swap	**PMX**	**Tournament Selection**	—	—
Swap	**PMX**	**Roulette Wheel**	—	—
————	————	————	————	————
Insertion	**Multi-Point**	**Stoic Tournament Selection**	30.40 segundos	17
Insertion	**Multi-Point**	**Tournament Selection**	31.27 segundos	17
Insertion	**Multi-Point**	**Roulette Wheel**	36.09 segundos	18
Insertion	**Single-Point**	**Stoic Tournament Selection**	7.29 segundos	18
Insertion	**Single-Point**	**Tournament Selection**	3.76 segundos	18
Insertion	**Single-Point**	**Roulette Wheel**	5.53 segundos	18
Insertion	**PMX**	**Stoic Tournament Selection**	—	—

Mutação	Cruzamento	Seleção	Tempo	Número de Contêineres
Insertion	**PMX**	**Tournament Selection**	—	—
Insertion	**PMX**	**Roulette Wheel**	—	—
————————	————————	————————	————————	————————
Inversion	**Multi-Point**	**Stoic Tournament Selection**	24.95 segundos	18
Inversion	**Multi-Point**	**Tournament Selection**	30.29 segundos	18
Inversion	**Multi-Point**	**Roulette Wheel**	26.13 segundos	18
Inversion	**Single-Point**	**Stoic Tournament Selection**	4.99 segundos	18
Inversion	**Single-Point**	**Tournament Selection**	9.42 segundos	18
Inversion	**Single-Point**	**Roulette Wheel**	19.21 segundos	18
Inversion	**PMX**	**Stoic Tournament Selection**	—	—
Inversion	**PMX**	**Tournament Selection**	—	—
Inversion	**PMX**	**Roulette Wheel**	—	—
————————	————————	————————	————————	————————
Bit Flip	**Multi-Point**	**Stoic Tournament Selection**	20.15 segundos	18
Bit Flip	**Multi-Point**	**Tournament Selection**	15.51 segundos	18
Bit Flip	**Multi-Point**	**Roulette Wheel**	19.49 segundos	18
Bit Flip	**Single-Point**	**Stoic Tournament Selection**	20.58 segundos	18
Bit Flip	**Single-Point**	**Tournament Selection**	3.23 segundos	18
Bit Flip	**Single-Point**	**Roulette Wheel**	19.76 segundos	18
Bit Flip	**PMX**	**Stoic Tournament Selection**	—	—
Bit Flip	**PMX**	**Tournament Selection**	—	—
Bit Flip	**PMX**	**Roulette Wheel**	—	—

4. Referencias

Alvim, A.C., Ribeiro, C.C., Glover, F. *et al.* A Hybrid Improvement Heuristic for the One-Dimensional Bin Packing Problem. *Journal of Heuristics* **10**, 205–229 (2004). https://doi.org/10.1023/B:HEUR.0000026267.44673.ed

Layeb, Abdesslem & Chenche, Sara. (2012). A Novel GRASP Algorithm for Solving the Bin Packing Problem. International Journal of Information Engineering and Electronic Business. 4. 8-14. 10.5815/ijieeb.2012.02.02.

F. Luo, I. D. Scherson and J. Fuentes, "A Novel Genetic Algorithm for Bin Packing Problem in jMetal," 2017 IEEE International Conference on Cognitive Computing (ICCC), Honolulu, HI, USA, 2017, pp. 17-23, doi: 10.1109/IEEE.ICCC.2017.10. keywords: {Biological cells;Genetic algorithms;Algorithm design and analysis;Heuristic algorithms;Optimization;Genetics;Evolutionary computation;bin packing;jMetal;genetic algorithm;optimization},

Ozcan, Sukru Ozer et al. “A Novel Grouping Genetic Algorithm for the One-Dimensional Bin Packing Problem on GPU.” *International Symposium on Computer and Information Sciences* (2016).

Quiroz-Castellanos, Marcela et al. “A grouping genetic algorithm with controlled gene transmission for the bin packing problem.” *Comput. Oper. Res.* 55 (2015): 52-64.

Borgulya, István. “A hybrid evolutionary algorithm for the offline Bin Packing Problem.” *Central European Journal of Operations Research* 29 (2020): 425 - 445.

Falkenauer, Emanuel. “A hybrid grouping genetic algorithm for bin packing.” *Journal of Heuristics* 2 (1996): 5-30.

Iima, Hitoshi and Tetsuya Yakawa. “A new design of genetic algorithm for bin packing.” *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* 2 (2003): 1044-1049 Vol.2.

He, Kun et al. “Adaptive large neighborhood search for solving the circle bin packing problem.” *Comput. Oper. Res.* 127 (2021): 105140.

Abdul-Minaam, Diaa Salama et al. “An Adaptive Fitness-Dependent Optimizer for the One-Dimensional Bin Packing Problem.” *IEEE Access* 8 (2020): 97959-97974.

Cardoso Silva, Aluísio and Carlos Cristiano Hasenclever Borges. “An Improved Heuristic Based Genetic Algorithm for Bin Packing Problem.” *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)* (2019): 60-65.

Scholl, Armin et al. “Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem.” *Comput. Oper. Res.* 24 (1997): 627-645.

Kucukyilmaz, Tayfun and Hakan Ezgi Kiziloz. “Cooperative parallel grouping genetic algorithm for the one-dimensional bin packing problem.” *Comput. Ind. Eng.* 125 (2018): 157-170.

Stawowy, Adam. “Evolutionary based heuristic for bin packing problem.” *Comput. Ind. Eng.* 55 (2008): 465-474.

Potarusov, Roman et al. “Hybrid genetic approach for 1-D bin packing problem.” *International Journal of Services Operations and Informatics* 6 (2011): 71.

Munien, Chanaleä et al. “Metaheuristic Approaches for One-Dimensional Bin Packing Problem: A Comparative Performance Study.” *IEEE Access* 8 (2020): 227438-227465.

Kaaouache, Mohamed Amine and Sadok Bouamama. “Solving bin Packing Problem with a Hybrid Genetic Algorithm for VM Placement in Cloud.” *International Conference on Knowledge-Based Intelligent Information & Engineering Systems* (2015).

Chan, Felix T. S. et al. “Using genetic algorithms to solve quality-related bin packing problem.” *Robotics and Computer-integrated Manufacturing* 23 (2007): 71-81.

5. Anexos

5.1. Sobre os Conjuntos de Dados de Benchmark e Avaliação de Desempenho

Em muitos artigos abordados, vários conjuntos de dados de benchmark padrão foram usados para avaliar e comparar o desempenho de diferentes algoritmos BPP. Alguns desses conjuntos de dados incluem aqueles introduzidos por Falkenauer, que são categorizados em:

- **Uniforme (u):** Instâncias com pesos de itens distribuídos uniformemente e uma capacidade de caixa fixa.
- **Tripletos (t):** Instâncias projetadas de forma que a solução ideal envolva empacotar três itens por caixa.
- **Difícil:** Instâncias que geralmente são mais desafiadoras para resolver de forma ideal.

A avaliação do desempenho do algoritmo geralmente envolve considerar:

- **Qualidade da Solução:** Medida por métricas como o número de caixas usadas ou a quantidade de espaço desperdiçado.
- **Tempo Computacional:** O tempo necessário para encontrar uma solução.
- **Escalabilidade:** Quão bem o algoritmo se comporta à medida que o tamanho do problema aumenta.

5.2. Operadores Implementados para a GGA

obs.: Foi colocada como caixas de marcadores para facilitar também a nossa visualização do que gostariamos de implementar e do que foi implementado, assim, facilitando também uma organização do código.

5.2.1. Operadores de Seleção:

- ☑ **Seleção por Roleta (Roulette Wheel Selection)**
- ☑ **Seleção por Torneio (Tournament Selection)**
- ☐ **Seleção por Classificação (Rank Selection)**
- ☐ **Seleção Estocástica Universal (Stochastic Universal Sampling)**
- ☐ **Seleção Truncada (Truncation Selection)**
- ☑ Stoic Tournament Selection

5.2.2. Operadores de Cruzamento (Recombinação)

- ☑ **Cruzamento de Ponto Único (Single-Point Crossover)**
- ☑ **Cruzamento de Múltiplos Pontos (Multi-Point Crossover)**
- ☐ **Cruzamento Uniforme (Uniform Crossover)**
- ☐ **Cruzamento Aritmético (Arithmetic Crossover)**
- ☑ **Cruzamento PMX (Partially Matched Crossover)**
- ☐ **Cruzamento de Ordem (Order Crossover - OX)**
- ☐ **Cruzamento Cíclico (Cycle Crossover - CX)**

5.2.3. Operadores de Mutação:

Para manter a diversidade genética.

- ☑ **Mutação de Bit Flip (Bit Flip Mutation)**
- ☑ **Mutação de Troca (Swap Mutation)**
- ☑ **Mutação de Inversão (Inversion Mutation)**
- ☑ **Mutação de Inserção (Insertion Mutation):**
- ☑ **Mutação de Scramble (Scramble Mutation):**
- ☑ **Mutação Gaussiana (Gaussian Mutation):**

5.2.4. Operadores de Reposição (Survivor Selection):

- [ ] **Reposição Geracional Completa (Generational Replacement):**
  - Toda a população é substituída pelos novos indivíduos.
  - Simples, mas pode levar à perda de boas soluções.
- [ ] **Reposição Parcial (Steady-State Replacement):**
  - Apenas alguns indivíduos são substituídos a cada geração.
  - Mantém parte da população anterior, preservando boas soluções.
- [x] **Elitismo:**
  - Garante que os melhores indivíduos da geração atual sejam mantidos na próxima geração.
  - Evita a perda de soluções de alta qualidade.

5.2.5. Operadores de Diversificação e Intensificação:

- [ ] **Diversificação:**
  - Introduz novas informações genéticas na população.
  - Evita a convergência prematura para ótimos locais.
  - Pode ser feito através de mutações mais agressivas ou reinicialização parcial da população.
- [x] **Intensificação:**
  - Foca na exploração profunda das áreas promissoras do espaço de busca.
  - Pode incorporar técnicas como busca local ou tabu search para melhorar soluções específicas.
