COMPUTAÇÃO EVOLUCIONÁRIA

RELATÓRIO 1
Algoritmo Genético para o Problema da Mochila.

Michelle Hanne Soares de Andrade

Belo Horizonte
Maio, 2025.

SUMÁRIO

1- INTRODUÇÃO

O problema da mochila é um clássico desafio da otimização combinatória que tem sido amplamente estudado desde sua primeira menção na literatura em Mathews (1897). Em 1957, George Dantzig propôs propôs um algoritmo guloso de aproximação para o problema da mochila ilimitada, que consiste em ordenar os itens em ordem decrescente de valor por unidade de peso (DANTZIG, 1957).

Uma das variações do problema da mochila mais comum a ser resolvido é o problema da mochila 0-1, que restringe o número x_i de cópias de cada tipo de item para zero ou um (KELLERER, 2004). Dado um conjunto de n itens numerados de 1 até n, cada um com um peso w_i e um valor v_i, juntamente com uma capacidade máxima de peso W. O problema consiste em maximizar a soma dos valores dos itens na mochila de modo que a soma dos pesos seja menor ou igual à capacidade da mochila:

maximizar $\sum_{i=1}^{n} v_i x_i$

sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$ e $x_i \in \{0, 1\}$.

Onde x_i representa o número de instâncias do item i para incluir na mochila.

Muitos estudos apresentam soluções sobre a discussão de possíveis soluções do problema da mochila com Computação Evolucionária, como Djannaty et al (2008), Pradhan et al, (2014), Bole et al (2017), Montazeri et al (2017).

Este relatório aborda uma solução do problema da mochila implementado em Algoritmo Genético.

2- SOLUÇÃO PARA O PROBLEMA DA MOCHILA

Este Algoritmo Genético foi projetado para resolver o problema da mochila, que consiste em selecionar um conjunto de itens – cada um com um peso e um valor específicos – de forma a maximizar o **valor total** sem exceder a capacidade de peso da mochila. A seguir, as principais explicações sobre as funções:

***Item*:** Esta classe representa cada item disponível, armazenando seu nome, peso e valor.

```
# Representa um item com peso e valor

class Item:
def __init__(self, nome, peso, valor):
self.nome = nome
self.peso = peso
self.valor = valor
```

Individuo: Esta classe representa uma solução candidata (uma possível combinação de itens na mochila).

```
# Representa um indivíduo (solução candidata)

class Individuo:
def __init__(self, itens, capacidade, genes=None):
self.itens = itens
self.capacidade = capacidade
if genes:
self.genes = genes
else:
self.genes = self._gerar_genes_validos()
self.avaliar_fitness()
```

_gerar_genes_validos(): Inicializa um indivíduo garantindo que a combinação de itens não exceda a capacidade da mochila. Os genes são uma lista de 0s e 1s, onde 1 significa que o item correspondente está na mochila e 0 que não está.

```
def _gerar_genes_validos(self):
# vetor das variáveis
genes = [0] * len(self.itens)
indices = list(range(len(self.itens)))
random.shuffle(indices)
peso_total = 0

for i in indices:
if peso_total + self.itens[i].peso <= self.capacidade:
genes[i] = 1
peso_total += self.itens[i].peso
```

```
return genes
```

avaliar_fitness(): Calcula a avaliação da *fitness* do indivíduo.

```
def avaliar_fitness(self):
# avaliar só quem precisa avaliar
self.peso_total = sum(item.peso for item, g in zip(self.itens,
self.genes) if g)
self.valor_total = sum(item.valor for item, g in zip(self.itens,
self.genes) if g)
self.fitness = self.valor_total if self.peso_total <= self.capacidade
else 0
```

crossover(): Realiza o cruzamento entre dois indivíduos (pais) para gerar dois novos indivíduos (filhos), combinando seus genes. O ponto de corte para o *crossover* é escolhido aleatoriamente.

```
# taxa de cruzamento por casal
def crossover(self, outro):
# Escolhe aleatoriamente um ponto de corte entre 1 e o tamanho dos
genes - 1.
ponto = random.randint(1, len(self.genes) - 1)

# Cria o primeiro filho, pegando os genes do início do self até o ponto
de corte e depois os genes do outro indivíduo (outro) do ponto até o
final.
filho1_genes = self.genes[:ponto] + outro.genes[ponto:]

# Cria o segundo filho, pegando os genes do início do outro até o ponto
de corte e depois os genes do self do ponto até o final.
filho2_genes = outro.genes[:ponto] + self.genes[ponto:]

return Individuo(self.itens, self.capacidade, filho1_genes),
Individuo(self.itens, self.capacidade, filho2_genes)
```

mutar(): Introduz pequenas alterações aleatórias nos genes de um indivíduo, com base em uma *taxa_mutacao*. Isso ajuda a manter a diversidade genética na população.

```
# a taxa de mutação influencia a mutação de cada individuo
def mutar(self, taxa_mutacao):
novos_genes = [1 - g if random.random() < taxa_mutacao else g for g in
self.genes]
self.genes = novos_genes
self.avaliar_fitness()
```

corrigir(): Garante que um indivíduo que excedeu a capacidade da mochila após o crossover ou mutação seja reparado, removendo itens aleatoriamente até que o peso total esteja dentro do limite.

```
def corrigir(self):
while self.peso_total > self.capacidade:
ativos = [i for i, g in enumerate(self.genes) if g == 1]
if not ativos:
break
i = random.choice(ativos)
self.genes[i] = 0
self.avaliar_fitness()
```

Populacao: Esta classe gerencia uma coleção de indivíduos com alguns métodos.

```
# codificar no formato de binário são 40 variaveis
def __init__(self, tamanho, itens, capacidade):
self.itens = itens
self.capacidade = capacidade
self.tamanho = tamanho
self.individuos = [Individuo(itens, capacidade) for _ in
range(tamanho)]
```

selecao(): Seleciona os dois indivíduos mais aptos da população para serem pais da próxima geração. Prioriza indivíduos válidos (que não excedem a capacidade).

```
def selecao(self):
candidatos_validos = [ind for ind in self.individuos if ind.peso_total
<= self.capacidade]
candidatos_validos = sorted(candidatos_validos, key=lambda ind:
ind.fitness, reverse=True)
return candidatos_validos[:2] if len(candidatos_validos) >= 2 else
sorted(self.individuos, key=lambda ind: ind.fitness, reverse=True)[:2]
```

`nova_geracao()`: Cria uma nova população. Ela pode incluir um número definido de "elites" (os melhores indivíduos da geração anterior) e o restante é gerado por seleção, **crossover** e **mutação** dos pais. A taxa de **crossover** (**`taxa_crossover`**) determina a probabilidade de os pais se cruzarem. A **taxa de mutação** é aplicada a cada filho gerado.

```
# Adicionado parâmetro para o número de elites
def nova_geracao(self, taxa_crossover=0.8, num_elites=1):
nova_pop = []
# Garante que o número de elites não exceda o tamanho da população
num_elites = min(num_elites, self.tamanho)

# Adiciona os indivíduos de elite à nova população
```

```
elite = sorted(self.individuos, key=lambda ind: ind.fitness,
reverse=True)[:num_elites]
nova_pop.extend(elite)

# Gera o restante da população através de seleção, crossover e mutação
while len(nova_pop) < self.tamanho:
pais = self.selecao()
# aplicar a taxa de cruzamento por casal
if random.random() < taxa_crossover: # Verifica se o crossover deve
ocorrer
filhos = pais[0].crossover(pais[1])
else: # Se não houver crossover, os filhos são cópias dos pais
filhos = (Individuo(self.itens, self.capacidade, list(pais[0].genes)),
Individuo(self.itens, self.capacidade, list(pais[1].genes)))

for filho in filhos:
filho.mutar(taxa_mutacao=0.2)
filho.corrigir()
nova_pop.append(filho)
if len(nova_pop) >= self.tamanho:
break
# criterios para a escolha do melhor individuo
self.individuos = sorted(nova_pop, key=lambda ind: ind.fitness,
reverse=True)[:self.tamanho]
```

melhor_individuo(): Retorna o indivíduo com o maior fitness na população atual.

```
def melhor_individuo(self):
return max(self.individuos, key=lambda ind: ind.fitness)
```

algoritmo_genetico(): É a função principal que executa o algoritmo. Ela inicializa a população e, a cada geração, cria uma nova população, acompanha o melhor indivíduo e registra o histórico de *fitness*.

```
# Adicionado parâmetro para o número de elites
def algoritmo_genetico(caminho_arquivo, tam_populacao=50,
max_geracoes=10, taxa_crossover=0.8, num_elites=1):
random.seed()
capacidade, itens = carregar_dados(caminho_arquivo)
populacao = Populacao(tam_populacao, itens, capacidade)

melhor_fitness = 0
geracoes_sem_melhora = 0
historico = []
```

```
for geracao in range(max_geracoes):
# Passa a taxa de crossover e o número de elites para nova_geracao
populacao.nova_geracao(taxa_crossover, num_elites)
melhor = populacao.melhor_individuo()

historico.append([geracao + 1, melhor.fitness, melhor.peso_total])

print(f"Geração {geracao + 1}: Melhor valor = {melhor.fitness}, Peso =
{melhor.peso_total}")

# Retorna o melhor indivíduo e o histórico
return populacao.melhor_individuo(), historico
```

2.1 – Função *Fitness*

A função *fitness*, implementada no método **avaliar_fitness()** da classe Individuo, determina a qualidade de uma solução candidata. Neste contexto, o *fitness* é o valor **total dos itens selecionados no indivíduo**. No entanto, se o **peso total** dos itens selecionados exceder a capacidade da mochila, o *fitness* do indivíduo é penalizado e definido como 0. Isso garante que apenas soluções válidas (que respeitam a restrição de capacidade) sejam consideradas boas.

2.2 – Principais Parâmetros do Algoritmo Genético

O comportamento e o desempenho deste algoritmo genético são influenciados por vários parâmetros, como a seguir:

- **`tam_populacao`**: O número de indivíduos em cada geração. Por exemplo: 50.
- **`max_geracoes`**: O número máximo de gerações que o algoritmo executará. Por exemplo: 10.
- **`taxa_crossover`**: A probabilidade de dois pais selecionados realizarem o cruzamento para gerar filhos. No código, é passada para a função *`nova_geracao`* e usada para decidir se o *crossover* ocorre para um casal. Por exemplo, `0.8` é usado na função *`main`*.
- **`taxa_mutacao`**: A probabilidade de cada gene de um indivíduo sofrer mutação. No código, durante a criação da nova geração, a função *`mutar`* é chamada com uma *`taxa_mutacao`*, `exemplo 0.2` (20%) para cada filho.
- **`num_elites`**: O número dos melhores indivíduos da geração atual que são diretamente transferidos para a próxima geração sem sofrer *crossover* ou mutação. Isso ajuda a preservar as melhores soluções encontradas. Na função `main`, *`num_elites_ag`* é definido como 1.
- **`capacidade`**: A capacidade máxima de peso da mochila, carregada do arquivo de dados.

A solução apresentada encontra-se no seguinte repositório do GitHub: https://github.com/mihanne/algoritmos_geneticos.

3- DISCUSSÃO DOS RESULTADOS

Foram executados **6 instâncias de dados** de diversos tamanhos, desde pequeno (40 itens) até grande (superior a 10.000 itens). Cada instância foi executada **30 vezes** para apuração dos resultados. O ambiente de execução foi o Google Colab[1], executando Python 3. Para instâncias pequenas foi utilizado a CPU Padrão (13 GB de RAM), para instâncias maiores que 10.000 itens foi configurado TPU v2-8 (dois núcleos – total 64 GB).

A seguir, a visão geral dos resultados de cada conjunto de dados

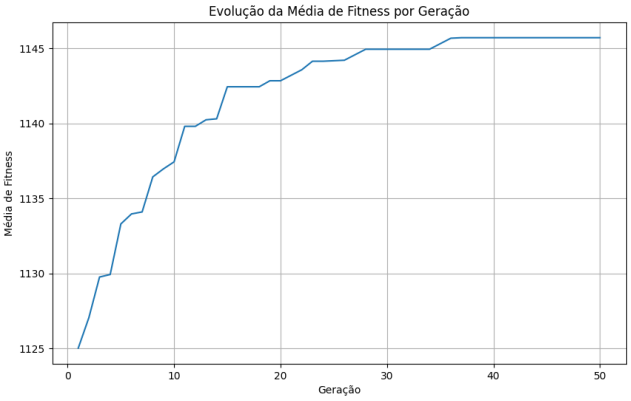
Instâncias	Peso da Mochila	Melhor Peso do AG[2]	Melhor Total do AG	Principais parâmetros	Tempo de Execução Total[3]
40 itens	15 Kg	15 Kg	1.149	tam_populacao=50 max_geracoes=50 taxa_crossover=0.8 num_elites=1 taxa_mutacao=0.2	6.97 seg.
100 itens	27 Kg	27 Kg	1.173	tam_populacao=50 max_geracoes=50 taxa_crossover=0.8 num_elites=1 taxa_mutacao=0.3	38,93 seg.
10.000 itens	431 Kg	431 Kg	4.851	tam_populacao=100 max_geracoes=100 taxa_crossover=0.8 taxa_mutacao_ind=0.2 num_elites=1	300,42 seg.
10.000 itens V2	1.765.326 Kg	1765326 Kg	784.051	tam_populacao=500 max_geracoes=500 taxa_crossover=0.8 taxa_mutacao_ind=0.10 num_elites=1	8.499,14 seg

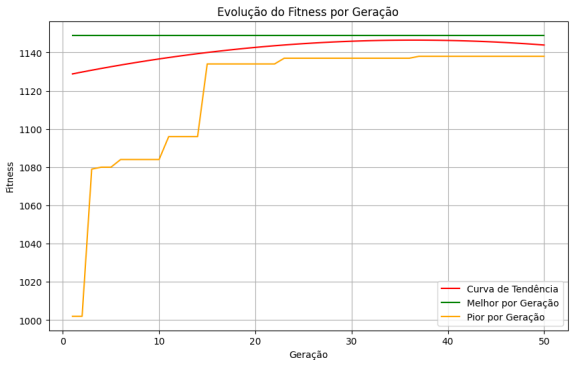
1 - https://colab.research.google.com/
2 - Algoritmo Genético
3 - Total de 30 execuções

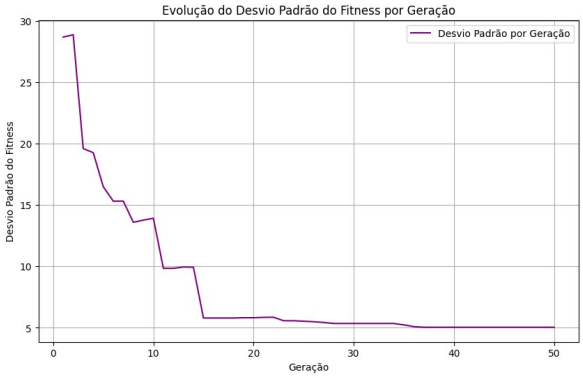
11.000 itens	1.000.000 Kg	999.944 Kg	456.053	tam_populacao=500 max_geracoes=500 taxa_crossover=0.8 taxa_mutacao_ind=0.1 num_elites=1	16.322,90 seg
		999.997 Kg	456.837	tam_populacao=1000 max_geracoes=50 taxa_crossover=0.8 taxa_mutacao_ind=0.01 num_elites=2	1.148,23 seg
100.000 itens	2.500.000 Kg	999.992 Kg	1.355.028	tam_populacao=100 max_geracoes=50 taxa_crossover=0.8 taxa_mutacao_ind=0.01 num_elites=2	3.915,82 seg

Instância de 40 itens

```
Melhor solução encontrada:
Valor total: 1149
Peso total: 15 de 15
Itens selecionados:
- X19 (peso=5, valor=381)
- X28 (peso=5, valor=386)
- X29 (peso=5, valor=382)
```

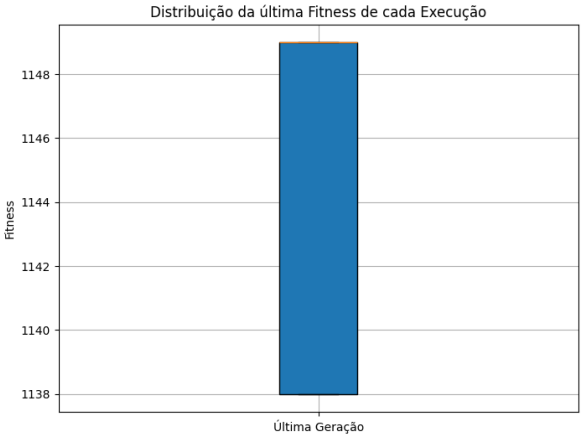






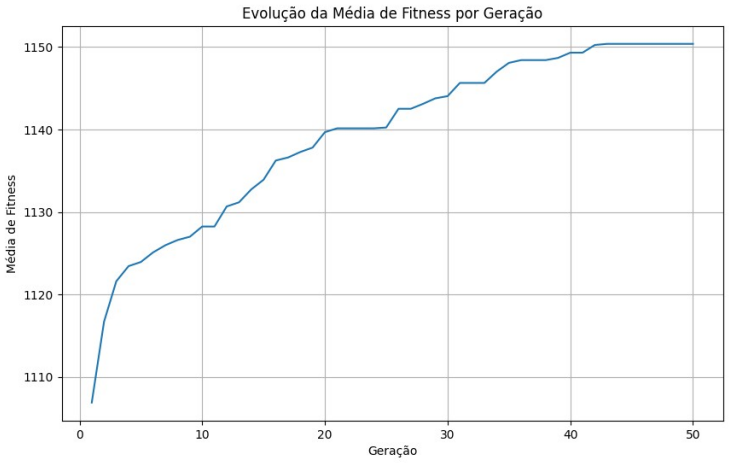
A melhor solução encontrada atingiu o *benchmark* estipulado como solução para esta instância do problema. Porém, percebe-se que a pior solução gerou indivíduos que iniciaram com valor total de 1000 até 1.138;

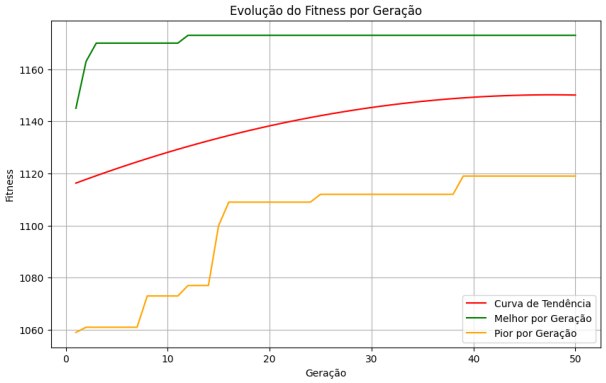
O desvio padrão da função *fitness* apresenta um declínio entre 29 até 5. Já a distribuição do último *fitness* de cada execução mostra que não há *outliers* e a mediana das execuções é de 1.149.

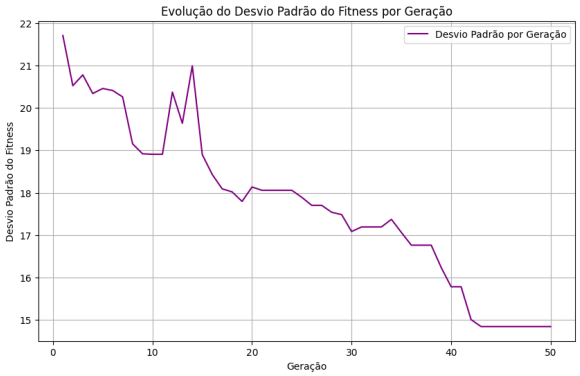


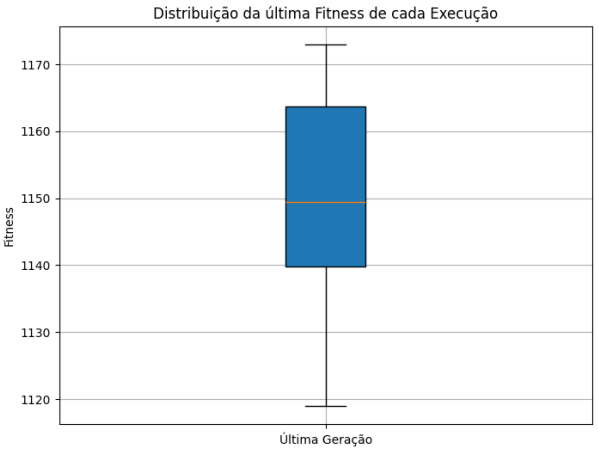
Instância de 100 itens

```
Melhor solução encontrada:
Valor total: 1173
Peso total: 27 de 27
Itens selecionados:
- X58 (peso=9, valor=384)
- X93 (peso=10, valor=390)
- X97 (peso=8, valor=399)
```





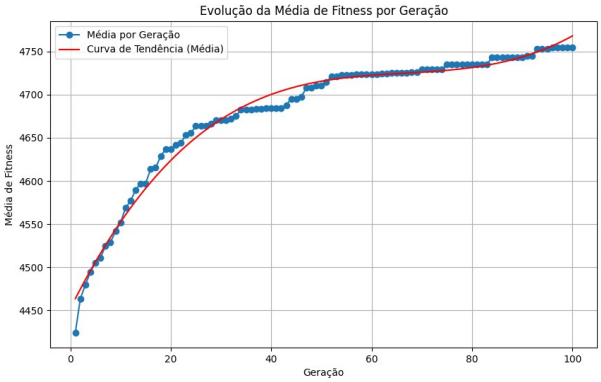




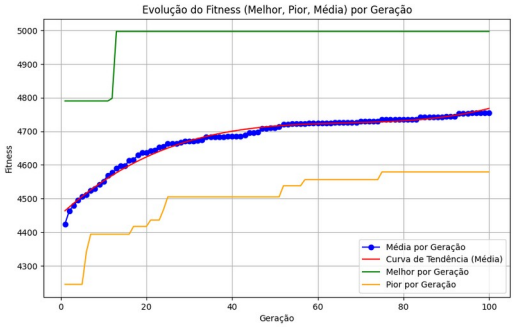
A melhor solução encontrada atingiu o *benchmark* estipulado como solução para esta instância do problema. Porém, percebe-se que a pior solução gerou indivíduos que iniciaram com valor total de 1060 até 1.120;

O desvio padrão da função *fitness* apresenta um declínio entre 22 até 15, porém, há momentos de aumento do desvio padrão entre as gerações 12 e 14. Já a distribuição do último *fitness* de cada execução mostra que não há *outliers* e a mediana das execuções é de 1.150.

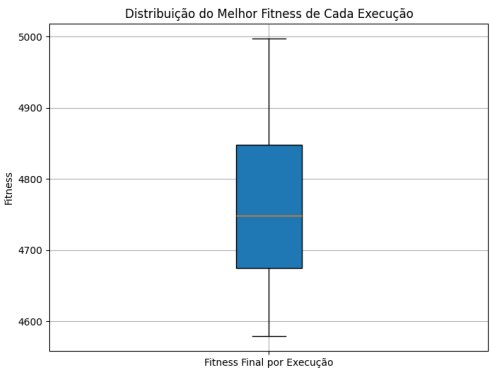
Instância de 10.000 itens



Melhor *fitness* geral encontrado: 4.997
Pior fitness (dos melhores de cada execução): 4.579
Média de *fitness* (dos melhores de cada execução): 4.754,87
Esta instância não atingiu o benchmark que é e 7.145.

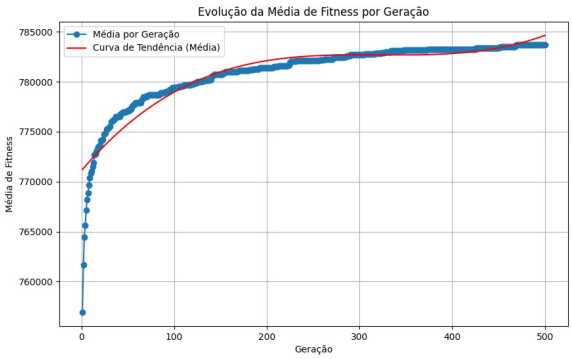


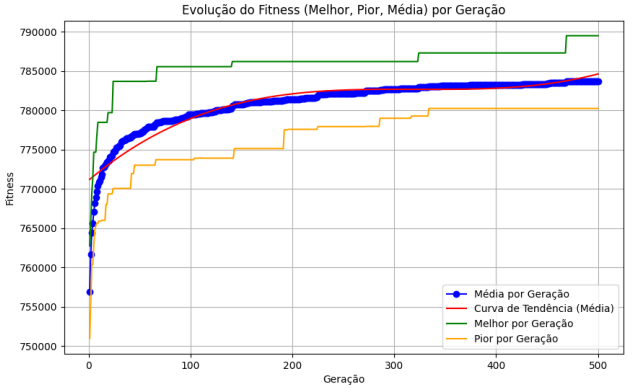
Nas instâncias grandes (igual ou superior a 10.000) foi gerada uma versão otimizada do algoritmo para que o tempo de execução fosse viável no Google Colab, com isso, a função *fitness* mesmo ajustada **não foi capaz de atingir o *benchmark* do problema (valor de 7.145).** O maior valor encontrado foi de 4.997, cujo diferença do *fitness* é de 2.148. O problema pode ser detectado na evolução do desvio padrão que oscila entre altos e baixos (grande variação). Já a mediana foi de aproximadamente 4.750.

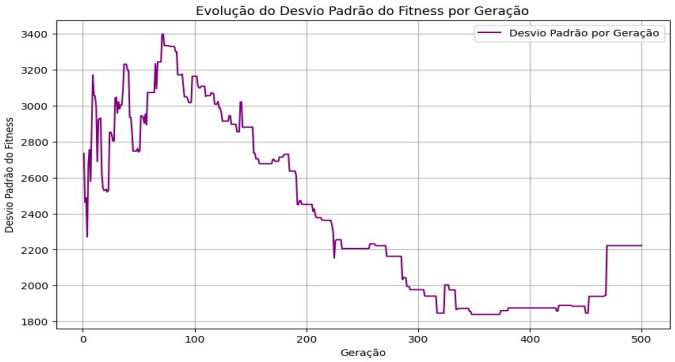


Instância de 10.000 itens v2

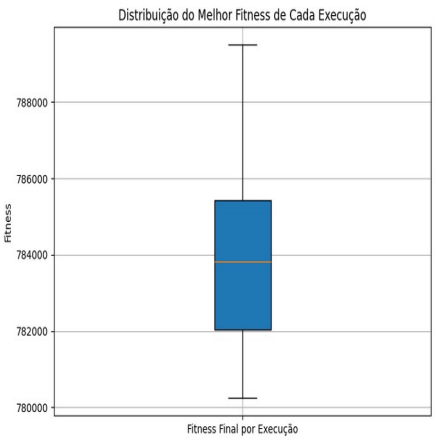
Melhor *fitness* geral encontrado: 789.501
Pior *fitness* (dos melhores de cada execução): 780.246
Média de *fitness* (dos melhores de cada execução): 783.698.03
Esta instância não atingiu o *benchmark* que é e 1.385.923.





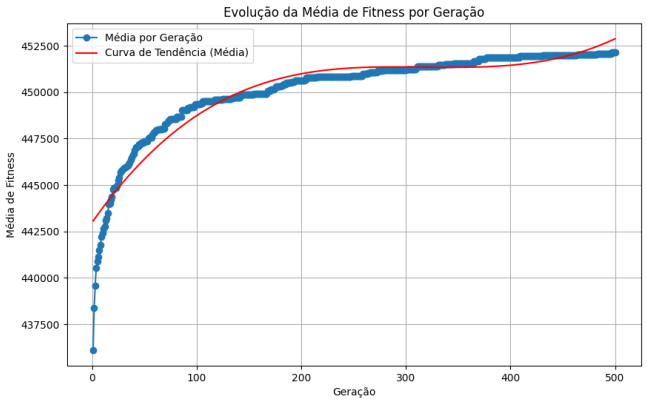


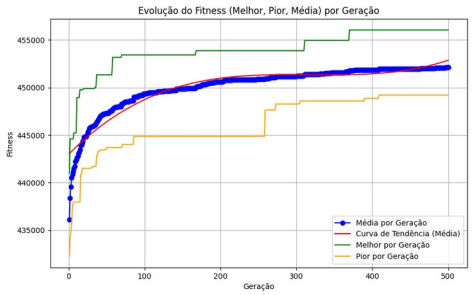
Nas instâncias grandes (igual ou superior a 10.000) foi gerada uma versão otimizada do algoritmo para que o tempo de execução fosse viável no Google Colab, com isso, a função *fitness* mesmo ajustada **não foi capaz de atingir o *benchmark* do problema (valor de 1.385.923).** O maior valor encontrado foi de 789.501, cujo diferença do *fitness* é de 587.422. O problema pode ser detectado na evolução do desvio padrão que oscila entre altos e baixos (grande variação). Já a mediana foi de aproximadamente 783.900.

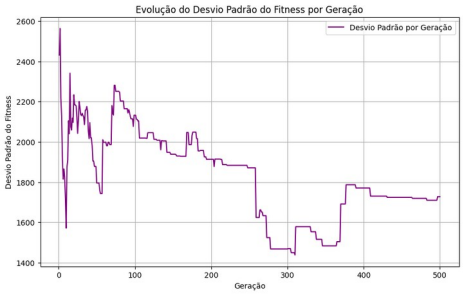


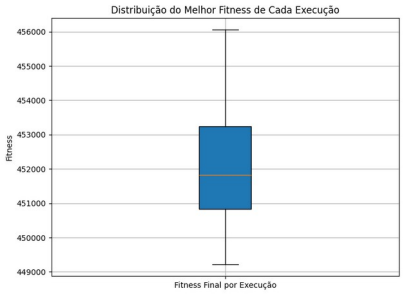
Instância de 11.000 itens

Melhor *fitness* geral encontrado: 456.053
Pior *fitness* (dos melhores de cada execução): 449.215
Média de *fitness* (dos melhores de cada execução): 452.135,83
Esta instância não atingiu o *benchmark* que é e 1.088.636, e ficou abaixo do peso da mochila de 1.000.000 Kg.



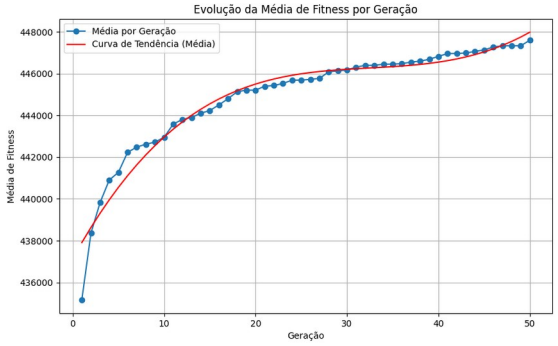


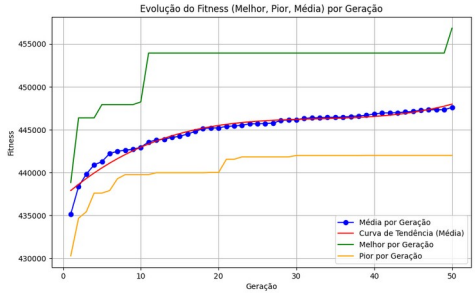


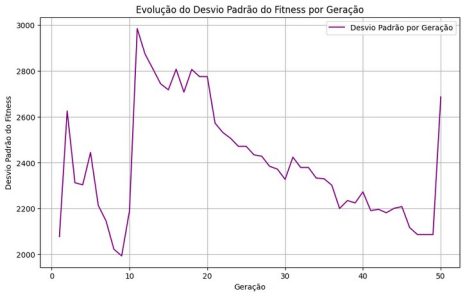


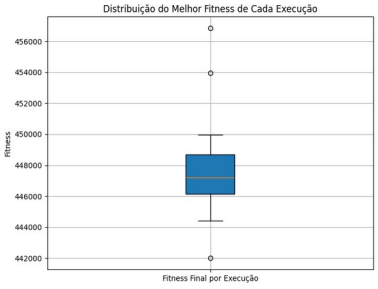
Nas instâncias grandes (igual ou superior a 10.000) foi gerada uma versão otimizada do algoritmo para que o tempo de execução fosse viável no Google Colab, com isso, a função *fitness* mesmo ajustada **não foi capaz de atingir o *benchmark* do problema (valor de 1.088.636), diferença de 632.583 dos valores dos itens.** Também não houve convergência para o peso da mochila, **diferença de 56 Kg.**
Neste problema foram executadas duas configurações de parâmetros diferentes, visando avaliar o **tamanho das gerações e da população.** No entanto, mesmo aumentando a população (**500**) e o número de gerações (**500**), o comportamento do algoritmo implementado não melhorou os resultados. O tempo de execução total foi de aproximadamente **4h30min.**
O desvio padrão teve grandes oscilações, mostrando que o comportamento do algoritmo não tende a uma solução ótima global e sim a ótimos locais, em alguns pontos.

Melhor *fitness* geral encontrado: 456.837
Pior *fitness* (dos melhores de cada execução): 442.005
Média de *fitness* (dos melhores de cada execução): 447.593,77
Esta instância não atingiu o *benchmark* que é e 1.088.636, e ficou abaixo do peso da mochila de 1.000.000 Kg.





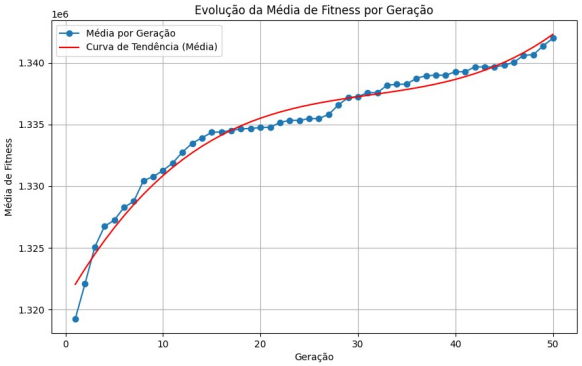


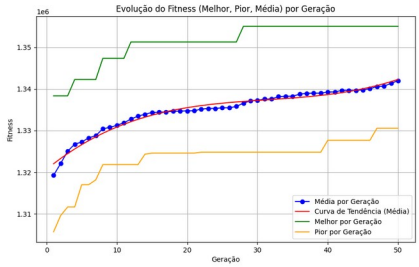


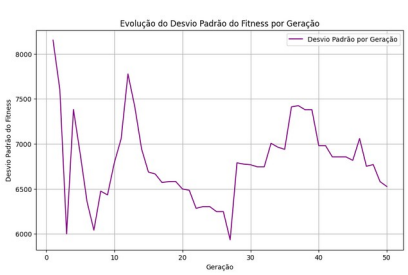
Os parâmetros foram alterados para verificar o impacto das gerações e da população. Reduziu-se as gerações para 50, aumentou a população para 1000, a taxa de mutação foi reduzida para 0,01 (1%) e o número de indivíduos selecionados no elitismo foi de 2. Com, isso, tentou-se preservar melhores indivíduos para as gerações seguintes, possibilitando gerar soluções progressivamente melhores. No entanto, os resultados apontam valores similares ao do primeiro teste com esta instância, o peso da mochila não atingiu o desejado, ficando **3 kg abaixo**. A função *fitness* não atingiu o seu ótimo, no melhor caso a diferença foi de **543.163** (valores dos itens).
O desvio padrão teve grandes oscilações, mostrando que o comportamento do algoritmo não tende a uma solução ótima global e sim a ótimos locais, em alguns pontos.
O gráfico da melhor solução *fitness* final de cada execução aponta os valores de máximo da mochila como *outliers*: [456837, 442005, 453935].

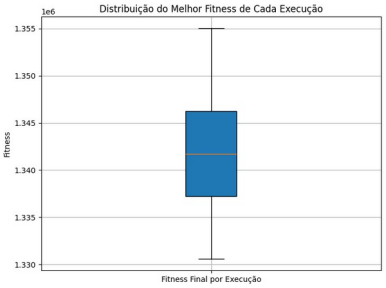
Instância de 100.000 itens

Melhor *fitness* geral encontrado: 1.355.028
Pior *fitness* (dos melhores de cada execução): 1.330.580
Média de *fitness* (dos melhores de cada execução): 1.341.981,17
Esta instância não atingiu o *benchmark* que é e 9.127.806, e ficou abaixo do peso da mochila de 2.500.000 Kg.









Os parâmetros foram reduzidos devido a questão de tempo. Percebe-se que o comportamento do algoritmo com número reduzido de gerações e população, atingiu aproximadamente apenas 40% do peso da mochila. Desse modo, a soma dos itens também esteve aquém do *fitness* esperado.
A execução do algoritmo com o número de **população de 10000** e **geração de 100**, já nas primeiras execuções mostrou-se mais promissora, atingindo um peso superior ao executado com os parâmetros acima, veja abaixo:

Valor total: 1.346.608
Peso total: 2.499.455 de 2500000

Porém, devido ao custo computacional a execução dessa configuração de parâmetros foi abortada.

4- CONCLUSÃO

A estratégia adotada em instâncias menores não funciona necessariamente para grandes instâncias. O número de gerações em instâncias maiores, por exemplo, entre 500 a 1000, impacta no tempo de execução, aumentando consideravelmente. O número da população também deverá ser maior, para preservar mais indivíduos para as gerações seguintes. A taxa de mutação deverá ser menor para construir soluções progressivamente melhores, evitando mudanças radicais. Por isso, foi adotado a taxa de mutação de 10% para as instâncias grandes, porém, esse valor poderia ser reduzido para 0,01 (1%) e 0,05 (5%) na tentativa de gerar soluções melhores que visassem o ótimo global.

A análise dos gráficos de desvio padrão das instâncias grandes aponta para uma convergência não estável. Em alguns casos, o algoritmo pode ter encontrando um "ótimo local", o desvio padrão começa a cair, mas pode vir a subir novamente. Nestes casos pode haver uma luta entre exploração (**buscar novas soluções**) e explotação (**refinar as soluções existentes**), sem que a explotação consiga dominar de forma consistente para levar a uma solução global.

A indicação para refinar o algoritmo, por exemplo:

- Detectar se o algoritmo estagnou, pois o melhor *fitness* da população não melhora. Quando detectada, poderá aumentar ***temporariamente a taxa de mutação***, por algumas gerações, de modo a “chacoalhar” a população e tirá-la do ótimo local.
- Reinicialização parcial da população, mantendo os melhores indivíduos (elitismo) e substituir o restante da população por novos indivíduos gerados aleatoriamente (ou por indivíduos fortemente mutados).

5- REFERÊNCIAS

BOLE, Amol V.; KUMAR, Rajesh. Multidimensional 0–1 Knapsack using directed Bee Colony algorithm. In: 2017 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS), 2017. p. 1-10.

DJANNATY, N.; DOUSTDARGHOLI, S.; TADAYON, F. A Hybrid Genetic Algorithm for the Multidimensional Knapsack Problem. In: INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY (ICCIT 2008), 2008, Khulna. Proceedings of the International Conference on Computer and Information Technology (ICCIT 2008). [S.l.: s.n.], 2008. p. 450-454.

KELLERER, Hans; PFERSCHY, Ulrich; PISINGER, David. Knapsack problems. Berlin: Springer, 2004. 461 p. ISBN.

MATHEWS, G. B. Sobre a partição dos números. Anais da Sociedade Matemática de Londres, Londres, v. 28, p. 486–490, 25 jun. 1897. doi: 10.1112/plms/s1-28.1.486.

MONTAZERI, Ojtaba; KIANI, Rasoul; RASTKHADIV, Seyed Saleh. A new approach to the Restart Genetic Algorithm to solve zero-one knapsack problem. In: 2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI), 2017. p. 0050-0053.

PRADHAN, Tribikram; ISRANI, Akash; SHARMA, Manish. Solving the 0–1 Knapsack problem using Genetic Algorithm and Rough Set Theory. In: 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, 2014. p. 1120-1125.