UNIVERSIDADE ESTADUAL DE MARINGÁ

UEM *CAMPUS* MARINGÁ

JOHN WILLIAM VICENTE, NAYANE BATISTA COSTA, YURI PIRES ALVES

RELATÓRIO SOBRE A APLICAÇÃO DE ALGORITMOS GENÉTICOS NO PROBLEMA DA MOCHILA BINÁRIA

MARINGÁ

2022

SUMÁRIO

1 INTRODUÇÃO

A definição teórica de uma metaheurística traz que, trata-se de uma estratégia de busca de um determinado problema, onde dentro dele, existem tentativas de exploração eficiente do espaço de suas soluções viáveis. São procedimentos e modelos gerais que guiam a construção de uma heurística.

Para explorarmos o conceito e o funcionamento das metaheurísticas na prática, confeccionamos o relatório a seguir a partir do desenvolvimento de uma solução para o Problema da Mochila Binária com a utilização do Algoritmo Genético e testes a partir de instâncias encontradas na *web*.

Ao modelarmos o algoritmo proposto, foram utilizados os princípios da orientação a objetos. Além disso, especificamos o uso de cinco operações específicas dentro do algoritmo genético implementado, sendo elas: população inicial, função *fitness*, *crossover*, mutação e critério de parada.

Após a apresentação da estrutura de dados aplicada, é mostrado os resultados obtidos a partir de cada versão testada, finalizando o trabalho proposto com as comparações e conclusões obtidas pela equipe após a análise das soluções.

2 O PROBLEMA DA MOCHILA BINÁRIA

2.0.1 Mochila binária

O problema da mochila binária consiste em encontrar uma combinação de itens, cada um com um valor *v* e um peso *p* específico, que não deve ultrapassar a capacidade *c* de uma mochila. A ideia por trás do algoritmo é encontrar os itens que maximize o maior valor possível e que possam ser levados pela mochila.

2.0.2 Instâncias do problema

Foi utilizado instâncias de testes para o problema da mochila encontradas no seguinte link: artemisa.unicauca. Neste endereço *web* foi encontrado diversas instâncias e os valores ótimos esperados para os algoritmos que resolvem esse problema.

Essas instâncias foram divididas em duas categorias: a de pequena dimensão, *Low-dimensional*, com a quantidade de itens menores e a de larga dimensão, *Large scale*, com problemas com uma quantidade maior de itens. Segue na figura 1 um exemplo da relação de arquivos e da resposta ótima encontrada no respectivo site.

10 test problems

file	Optimum
f1_l-d_kp_10_269	295
f2_l-d_kp_20_878	1024
f3_l-d_kp_4_20	35
f4_l-d_kp_4_11	23
f5_l-d_kp_15_375	481,0694
f6_l-d_kp_10_60	52
f7_l-d_kp_7_50	107
f8_l-d_kp_23_10000	9767
f9_l-d_kp_5_80	130
f10_l-d_kp_20_879	1025

Figura 1 – Instâncias do problema da mochila

O nome da instância já traz por si mesmo algumas informações relevantes a serem consideradas, por exemplo, os últimos dois valores separados pelo *'_'* simbolizam respectivamente o número de itens e a capacidade da mochila naquele caso. Ou seja, **f1_l-d_kp_10_269** simboliza uma instância de um problema com uma mochila de capacidade 269 e uma lista de 10 elementos.

3 DESENVOLVIMENTO DE UMA SOLUÇÃO COM ALGORITMOS GENÉTICOS

3.0.1 Modelagem do algoritmo

A heurística de algoritmos genéticos foi escolhida para solucionar o problema da mochila. Ela busca resolver problemas computacionais de otimização simulando o comportamento da seleção natural, encontrado na natureza e descrito primeiramente por Charles Darwin.

Nessa teoria, dentro de uma população de indivíduos há uma disputa de recursos e uma necessidade de sobrevivência em um determinado ambiente, onde existe uma tendência para que indivíduos mais aptos conseguem reproduzir-se deixando sua caracteristicas genética para as gerações futuras. Com o tempo as gerações de indivíduos começam a se especializar e se tornarem mais aptas em seu ambiente, seja através do recebimento do material genético de gerações pais quanto por mutações geradas de forma natural.

Os algoritmos genéticos tendem a ser uma heurística flexível quanto a forma como cada etapa será realizada para a emulação de um sistema de seleção de soluções mais aptas, porém a forma do algoritmo padrão dessa heurística possui alguns passos determinados, que são: população inicial, seleção de indivíduos, cruzamento e mutação e geração de novas populações. O fluxograma do algoritmo pode ser acompanhado na figura 2. O fluxo do programa se finaliza através de algum critério de parada, que pode ser diferente de acordo com o problema que o algoritmo resolve.

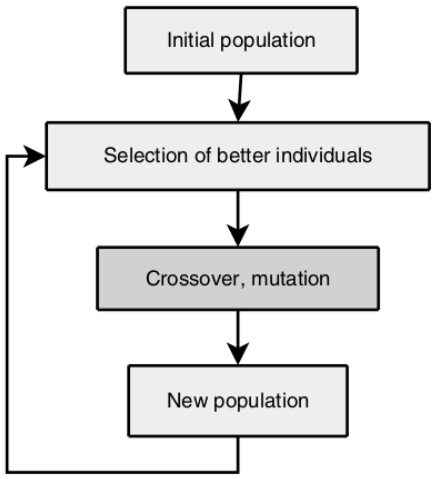

Figura 2 – Algoritmo genético

A solução para esse segundo trabalho foi implementada na linguagem Java. Essa linguagem foi escolhida devido ao seu acervo de bibliotecas, que já são bem consistentes, e por sua capacidade de ser uma linguagem multiplataforma.

Para a modelagem do algoritmo foi utilizado principios de orientação a objetos. Com esses princípios foi mais fácil realizar modificações no algoritmo e coletar dados provenientes dos testes realizados. A figura 3 representa o diagrama de classes do projeto.

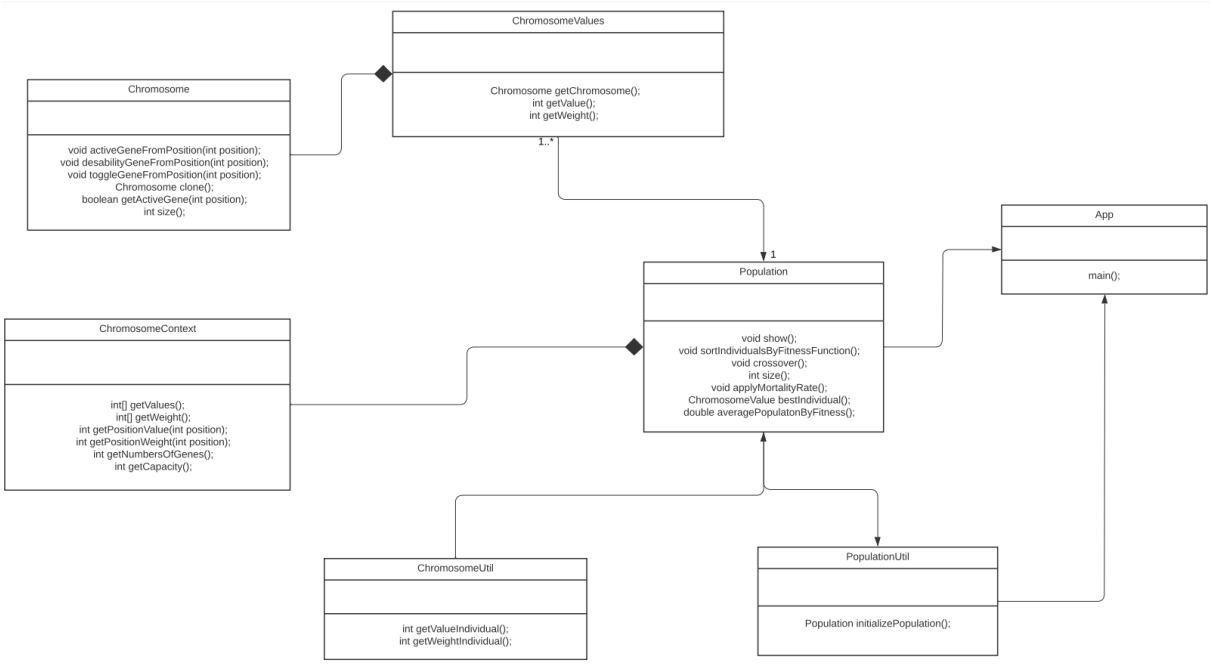

Figura 3 – Diagrama de classes - UML

3.0.2 Operações do algoritmo

Segue abaixo as operações junto de uma explicação das operações específicas do algoritmo genético implementadas no segundo trabalho prático da disciplina de modelagem e otimização algoritmica.

3.0.2.1 População inicial

A população inicial é um conjunto de indivíduos selecionados de alguma maneira, essa parte é fundamental para o algoritmo genético já que é nela que é fornecido o material com que as outras etapas irão realizar algum processamento. Cada indivíduo pode ser representado por um conjunto de dados que representa uma determinada carga genética.

Sobre a população inicial é realizado operações como seleção, cruzamento e descarte de indivíduos. No fluxo de dados do algoritmo novas populações podem ser geradas ou modificadas.

No trabalho supracitado, foram utilizadas duas formas de geração de uma população inicial. A primeira maneira escolhida foi a geração de uma população de forma aleatória, ou

seja foram escolhidos indivíduos com carga genética selecionada de forma aleatória. A segunda forma foi a tentativa de se gerar uma população mais orientada para a solução ótima, nela foi gerado um indivíduo através de uma heurística construtiva, levando em quando o valor relativo de cada gene, é selecionado indivíduos com uma certa variação genética a partir desse indivíduo gerado. Segundo a bibliografia consultada um bom tamanho para uma população gira em torno de 20 a 100 indivíduos (TALBI, 2009).

3.0.2.2 Função fitness

A função *fitness* tem como objetivo relacionar um indivíduo com uma espécie de valor quantitativo da sua adaptação ao determinado ambiente. Ela é interessante pois é possível medir o comportamento de uma determinada população através da média da função fitness selecionada, por exemplo, é possível visualizar se houve uma especialização da população para uma solução ótima com isso. Também é usada para auxiliar nas etapas de cruzamento, onde é dada maior probabilidade de selecionar indivíduos com maior potencial adaptativo.

A função *fitness* escolhida foi o valor de cada indivíduo, sendo cada indivíduo uma solução para o problema da mochila. Ou seja, é o valor que determinada combinação de itens, que respeitam a capacidade de uma mochila, possui. Essa função *fitness* foi utilizada para a ordenação de indivíduos, seleção e também auxiliou na operação de retirada de indivíduos menos aptos da população.

3.0.2.3 Crossover

A operações *crossover* ou cruzamento tem como objetivo gerar novos indivíduos que serão compostos dentro de uma nova população. Para isso, essa operação faz uso de indivíduos já presentes em uma determinada população.

Dentro da etapa de cruzamento existe também a parte de seleção, essa parte pode ser desenvolvida de várias formas possíveis, desde o método da seleção pela roleta, amostragem universal estocástica, por torneio e baseada em classificação.

Na etapa de geração de novos indivíduos é feita uma combinação dos genes. Existem muitas estratégias para essa operação, uma delas é combinar dois cromossomos pais gerando filhos a partir disso.

Na implementação desenvolvida foi implementado o modelo de seleção através do método da roleta. Nesse método, indivíduos com maior valor fitness têm mais possibilidade

de serem selecionados do que indivíduos com valores menores. Para a geração de filhos foi realizada a seguinte estratégia: definido um ponto de cruzamento os códigos de genes dos pais são quebrados nesse pontos e entrelaçados para geração de novos cromossomos, esses novos cromossomos são validados antes de serem transferidos para a população principal.

3.0.2.4 Mutação

Na etapa de cruzamento e geração de novos indivíduos é possível inserir mudanças de genes nos cromossomos filhos, a fim de aumentar a diversidade genética e trazer novas combinações possíveis para os genes.

Nessa parte realizamos uma determinada quantidade de mutações, a partir do tamanho da população, era realizado um sorteio para selecionar os filhos de um par de cromossomos pais. A partir desse sorteio é selecionado qual gene irá ser mudado, quando isso ocorre é realizado uma operação para validação do cromossomo e a sua inserção na população principal.

3.0.2.5 Critério de parada

O algoritmo genético não declara uma espécie de critério de parada padrão, quanto a isso a metaheurística é bastante flexível. Ou seja, determinando o problema abordado pode haver diferentes critérios de paradas.

A primeira forma adotada foi a designação manual de uma quantidade específica de gerações, iterações, que o algoritmo deveria realizar, ela foi útil para realizar testes, mas se viu um tanto inadequada, pois em alguns casos a quantidade de gerações impostas para o algoritmo não era o bastante para que se conseguisse uma resposta interessante.

Dessa maneira, para a segunda forma de parada foi percebido que em um determinado momento do processamento a média da função *fitness* da população não se alterava de maneira significativa, com isso foi definido o método de parada, que se a média fitness de uma nova população não variasse em mais de um por cento durante cinquenta gerações a população não conteria uma alta quantidade de variabilidade genética, a média das soluções ficaria oscilando em uma diferença muito pequena de geração para geração.

Esse método de parada pode ser um tanto arbitrário, porém se viu útil ao analisar os gráficos gerados pelo programa, como pode se ver na figura 4 que a média de *fitness* da população cresce até se estagnar e logo após isso o algoritmo se encerra.

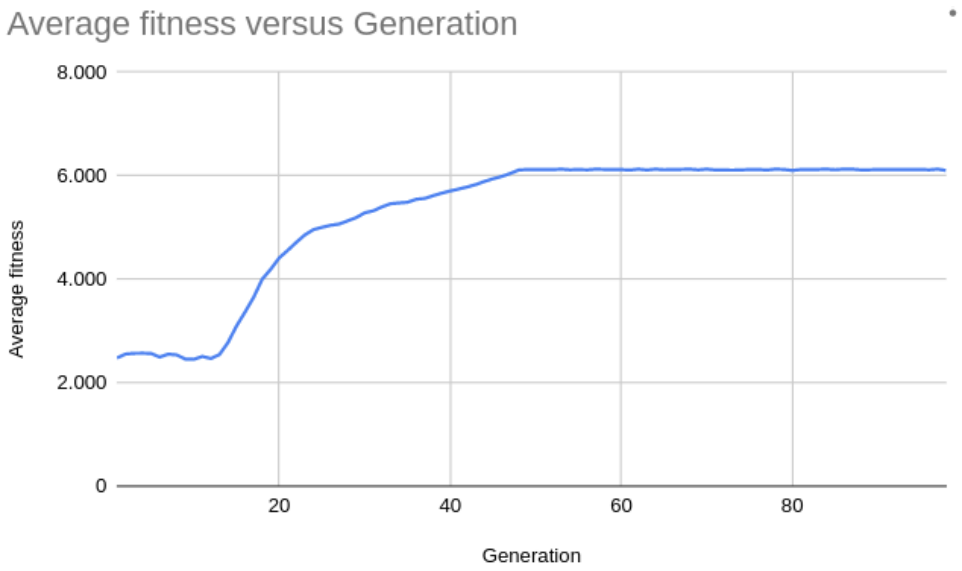

Figura 4 – Evolução dá média da função *fitness* durante o processamento do algoritmo

3.0.3 Estruturas de dados

Foram utilizadas algumas estruturas de dados para auxiliar e representar os dados envolvidos no algoritmo genético implementado. Como foi utilizado classes para representar abstrações, as estruturas de dados estão estão codificadas dentro delas, segue a seguir a especificação de algumas partes do algoritmo analisando as classes do projeto, encontradas na figura 3:

- ***Chromosome***: É uma classe que representa a configuração genética de um indivíduo. Para representar os genes ela utiliza um *array* de *booleans*. Cada posição do *array* pode ter o valor *True* ou *False* e isso indica a presença e falta de um gene no indivíduo respectivamente. Ele possui métodos para manipulação desse *array* como desativar ou ativar um gene, verificar se um gene está ativo ou não e por fim conseguir clonar o objeto e o *array* de genes de um determinado *Chromosome*.

- ***ChomosomeContext***: essa classe representa as instâncias de um determinado caso do problema da mochila. Ela abstrai os pesos e os valores dos itens que agora dentro da classe são designados como sendo genes que possuem um peso e um valor. Ela também possui um atributo para a quantidade de genes que existem e a capacidade máxima que um *Chromosome* pode ter. Ela é uma espécie de classe auxiliar que serve como consulta de informações da instância do problema envolvido. Ela utiliza *arrays* para receber o valores e também alguns atributos como a capacidade máximas.

- ***ChromosomeValue***: o valor e o peso de um conjunto de genes, cromossomos, pode variar

de ambiente para ambiente, esse contexto de ambiente pode ser relacionado com a classe *ChomosomeContext*, ela que representa os pesos e valores de cada gene. Para evitar o retrabalho de calcular muitas vezes o peso de cada indivíduo, que também pode ser considerado um *Chomosome*, é utilizado essa classe para armazenar o objeto *Chromosome* e o valor/peso respectivo. Ou seja, ela auxilia armazenando valores e servindo para consultas mais rápidas de dados importantes para os algoritmos.

- ***ChomosomeUtil*** e ***PopulationUtil***: São classes secundárias que apenas servem para fornecer métodos estáticos para auxiliar outras operações do sistema. Essas funções são para inicializar a População, da classe *PopulationUtil* e na classe *ChomosomeUtil* apresenta funções que relacionan um *Chromosome* a um *ChomosomeContext*, ou seja, ela serve para conseguir obter os valores e pesos dos cromossomos.

- ***Population***: É uma classe que apresenta o core do projeto. Ele possui as seguintes funções:

  - *show*: que mostra todos os indivíduos de uma população em determinada geração;
  - *sortIndividualsByFitnessFunction*: o objetivo dessa função é ordenar a população a partir do valor de cada indivíduo em determinado contexto. O valor de um indivíduo é possível de ser conseguido através das funções de *ChromosomeUtil*, que necessita de um *ChomosomeContext* para obter os dados dos genes(pesos/valores). A ordenação é feita através da biblioteca *Collection* da linguagem Java e a população é um conjunto de indivíduos(*Chromosome*) que estão dentro de uma lista;
  - *crossover*: O *crossover* reuniu algumas operações importantes. Em ordem, a primeira delas é a definição da quantidade de cruzamentos que irão acontecer, implementamos isso para que seja uma porcentagem da população como 10%, 15% e 50% da população. Após isso o algoritmo define a população de cruzamento, utilizamos o método de roleta que simula uma 'roleta' viciada que dá mais chances para aqueles indivíduos com maior valor da função *fitness*, cada interação da roleta conseguimos sortear um indivíduo que não participa mais da roleta em outras seleções, para isso acontecer a estrutura de dados utilizadas para a população de cruzamento foi o *Set* que removemos o indivíduo toda vez que ele é sorteado. Após isso ocorre o que chamamos de cruzamento, onde da lista da população de cruzamento são selecionados dois indivíduos, que têm seus genes meio que misturados, onde é sorteado um ponto aleatório no cromossomo que divide o cromossomo e é possível combinar diferentes

partes para formar indivíduos filhos que são diferentes dos seus pais, exemplificado na figura 5. Por fim, os indivíduos são avaliados se sua capacidade não extrapolar a da instância são colocados na população presente que formará a nova população da geração seguinte.

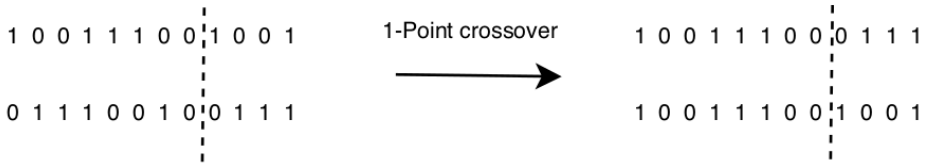

Figura 5 – Recombinação em um ponto

3.0.4 Versões e resultados

O projeto da implementação passou por alguns estágios e algumas decisões foram tomadas quanto a isso. nos próximos tópicos serão abordados as características, testes e os resultados obtidos ao submeter algumas instâncias de testes no programa desenvolvido.

3.0.4.1 v1.0

Na primeira versão é necessário configurar manualmente a quantidade de iterações que o algoritmo teria que realizar. Era inicializada uma quantidade de 100 indivíduos com carga genética formada de forma aleatória, foi colocado uma saída padrão que indica o tamanho da população antes e depois de uma iteração. A população é ordenada seguindo uma função *fitness*, que obtém o valor de indivíduo a partir da soma dos valores dos genes que estão ativos. Na parte de *crossover* é definido que a quantidade de cruzamentos seria de 20% da quantidade da população. A também a etapa de selecionar um indivíduo a partir do método da roleta, os indivíduos são salvos em uma estrutura de dados set e quando são selecionados também são removidos desse conjunto. No método de cruzamento são selecionados dois a dois indivíduos sorteados na etapa anterior e é aplicado o método de recombinação de genes descrito na parte de estrutura de dados em *crossover*. Logo após isso tem se a nova população. Também é retirado uma porcentagem dos menores valores presentes na população, essa porcentagem é de 6% e foi selecionada a partir de um dado de uma população de um país.

Ao executar o programa foi perceptível que a quantidade da população cresce exponencialmente, aumentando a quantidade de indivíduos bem rápida. Isso fez com que se tivesse um

maior processamento de dados para as tarefas de ordenação e cruzamento de dados, o programa ficou bem lento. Testamos a instância f1_l-d_kp_10_269 com 50 gerações e o programa conseguiu chegar na resposta ótima de 295, esse foi o valor do indivíduo com maior valor da função *fitness* encontrado na última geração. A última geração encontrava-se com um total de 15327 indivíduos. Para instâncias maiores seria necessário uma maior quantidade de iterações e isso faria com que a população aumentasse muito tornando o seu processamento mais demorado. Muitas das instâncias testadas não chegaram na resposta ótima, muitas das vezes nem chegaram perto dos valores ótimos. Por exemplo, ao testar a instância f2_l-d_kp_20_878 se obteve o valor 944 e uma segunda interação foi 860 sendo que seu ótimo é 1025. Para instâncias maiores foi inviável pois necessitavam de uma maior quantidade de iterações e isso se mostrou inviável.

3.0.4.2 v1.1

Ao ler um pouco da literatura no livro METAHEURISTICS (TALBI, 2009), foi sugerido a utilização de uma população menor de apenas 100 indivíduos. Inicializamos a população com apenas 20 indivíduos aleatórios. Também configuramos para que sempre, depois de adicionar os novos indivíduos, da parte de cruzamento e realizar a ordenação pela função fitness haveria um corte na quantidade de indivíduos para que esses não ultrapassassem a quantidade sugerida pela literatura.

Foi criado algumas funções para exportar os dados de uma população em um arquivo com extensão csv, nesse arquivo armazena dados de cada uma das iterações que o programa realizou dados como: número da iteração, tamanho da população, média *fitness* da população, melhor valor *fitness* encontrado e o peso do indivíduo com melhor *fitness*.

Foram realizados vários testes com as instâncias, pois cada interação poderia chegar em algum valor diferente já que o algoritmo usa princípios de aleatoriedade. Logo abaixo se encontra alguns resultados para as instâncias testadas:

Instância	pior resposta	melhor resposta	valor ótimo da instância
f1_l-d_kp_10_269	246	295	295
f2_l-d_kp_20_878	913	1024	1024
knapPI_1_100_1000_1	4371	5677	9147

Com os dados da população é possível montar um gráfico do comportamento da média *fitness* da população, segue na figura 6 e figura 7 um desses exemplos:

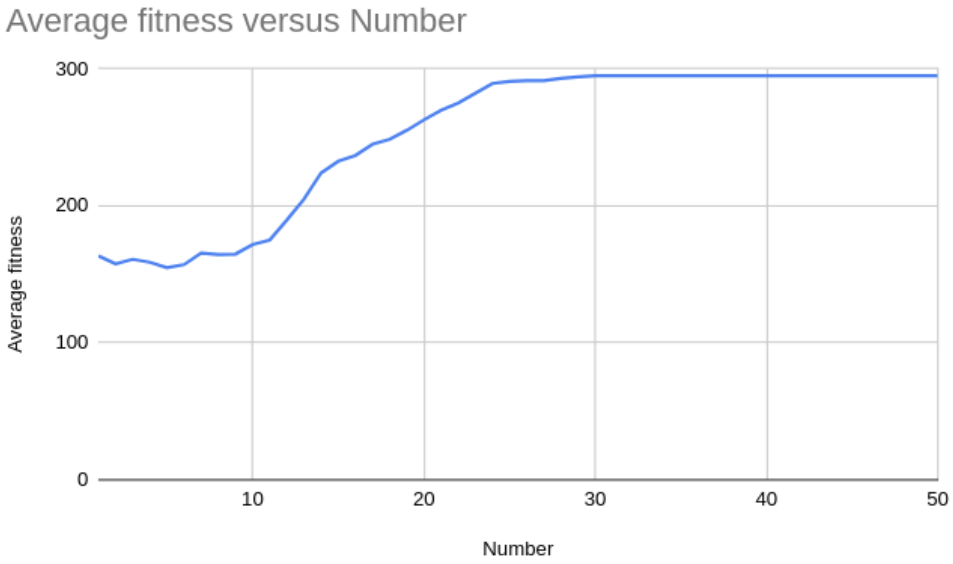

Figura 6 – melhor resposta da instância f1_l-d_kp_10_269

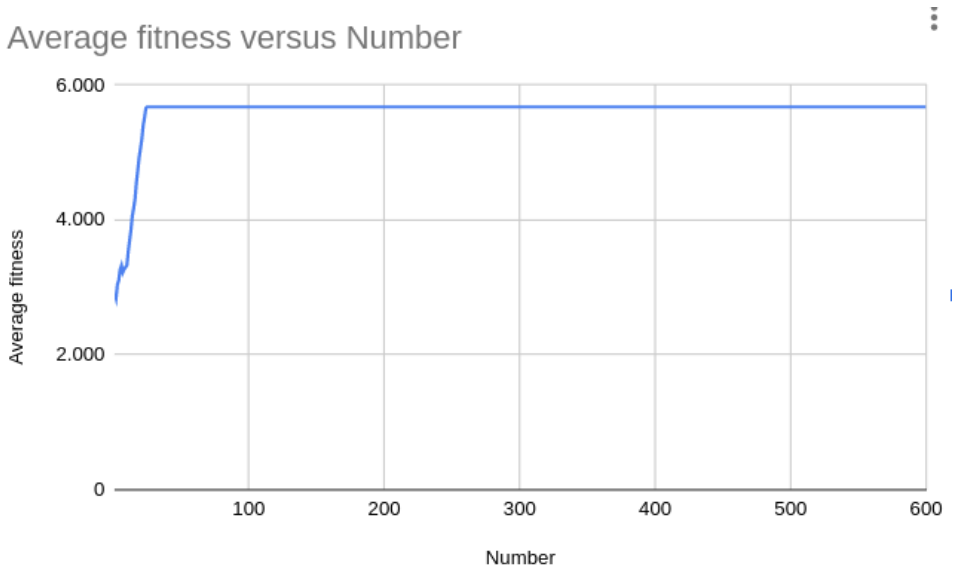

Figura 7 – melhor resposta da instância knapPI_1_100_1000_1

3.0.4.3 v1.3

Com os testes e a análise dos gráficos da versão foi percebido que a partir de um momento a variabilidade de uma população diminui ele e sua média de *fitness* também não modifica muito durante o tempo. Visto isso e a fim de se criar um método de parada foi criada a seguinte lógica: para cada iteração do algoritmo será calculada a média *fitness* da população e comparada com o valor da população anterior, se a diferença for menor ou igual a 1% é criado um contador que se incrementado por 50 iterações consecutivas irá finalizar o programa.

Também foi implementada a parte de mutação, onde uma determinada quantidade de indivíduos filhos têm algum de seus genes modificados pelo processo de mutação.

Instância	pior resposta	melhor resposta	valor ótimo da instância
f1_l-d_kp_10_269	294	295	295
f2_l-d_kp_20_878	995	1024	1024
f8_l-d_kp_23_10000	9748	9755	9767
knapPI_1_100_1000_1	5409	7707	9147

Segue abaixo alguns testes realizados:

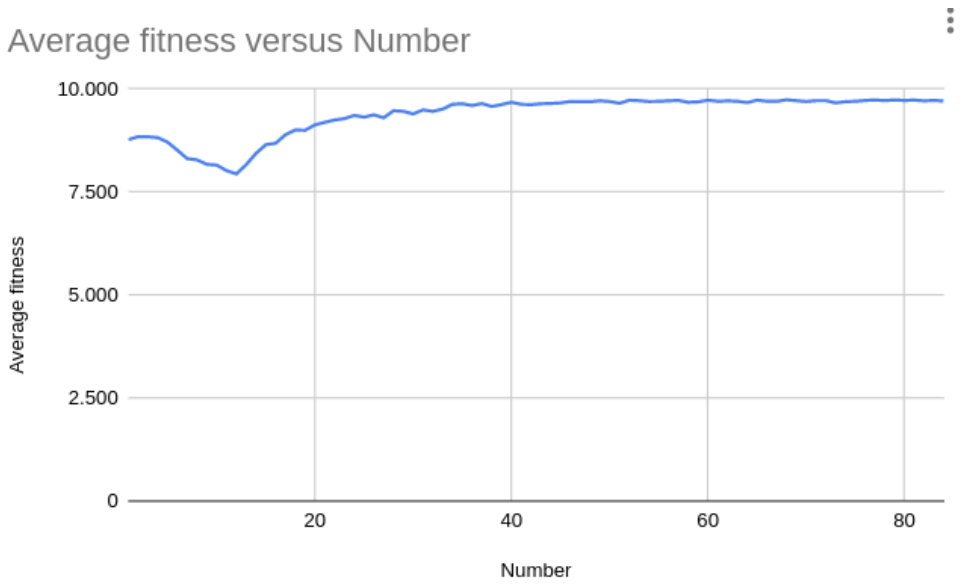

Figura 8 – melhor resposta da instância f8_l-d_kp_23_10000

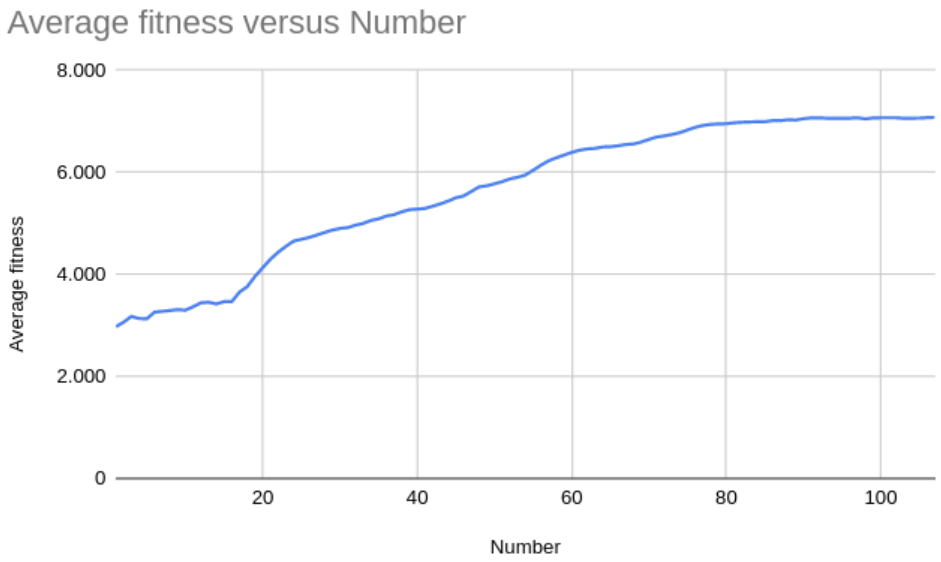

Figura 9 – melhor resposta da instância knapPI_1_100_1000_1

Esses detalhes implementados trouxe uma melhoria na qualidade das soluções, aparentemente o modelo de mutação criado surtiu um efeito positivo nas soluções geradas. Isso é possível pois aumenta as combinações genéticas, o que contribui para que indivíduos gerem filhos com

características diferentes e que sejam positivas no contexto da instância do problema. Para testes com uma quantidade pequena de ints/genes o algoritmo responde bem, porém com instâncias que ultrapassam os 100 itens/genes o algoritmo não obteve o mesmo desempenho.

3.0.4.4 v1.4

Percebendo o comportamento positivo de aumentar a variabilidade genética como a função de mutação, buscamos modificar a função de crossover para que apenas os 15% dos indivíduos com maior valor da função *fitness* com os 15% de menor valor. Objetivamos aumentar a possibilidade de combinações de genes combinando esses dois opostos. Segue abaixo a tabela com os testes realizados:

Instância	pior resposta	melhor resposta	valor ótimo da instância
f1_l-d_kp_10_269	295	295	295
f2_l-d_kp_20_878	946	1024	1024
f8_l-d_kp_23_10000	9759	9762	9767
knapPI_1_100_1000_1	5364	7718	9147

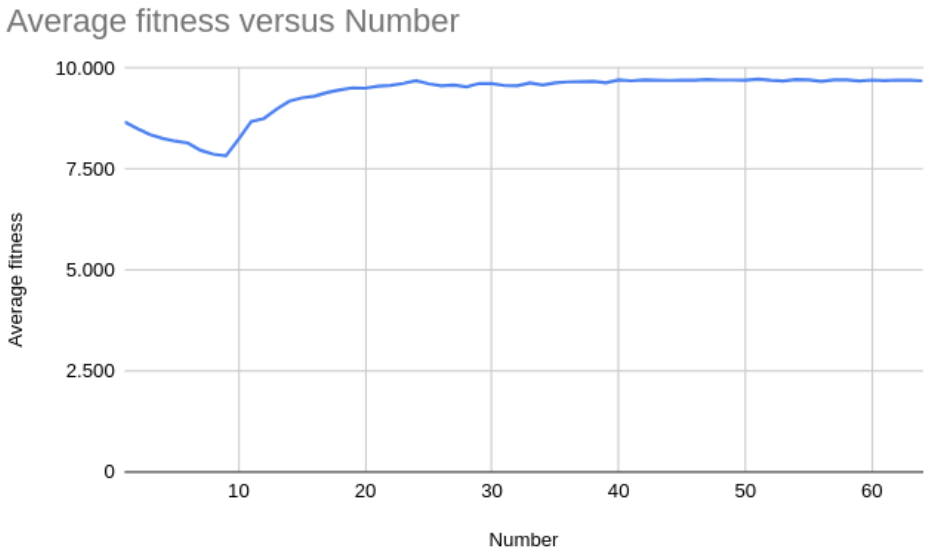

Figura 10 – melhor resposta da instância f8_l-d_kp_23_10000

Ao observar os resultados dos testes percebemos que a modificação realizada não surtiu um efeito esperado, esperávamos que a média populacional chegasse mais perto da ótima solução, mas de maneira geral ela continuou devolvendo valores semelhantes aos de antes dessa versão.

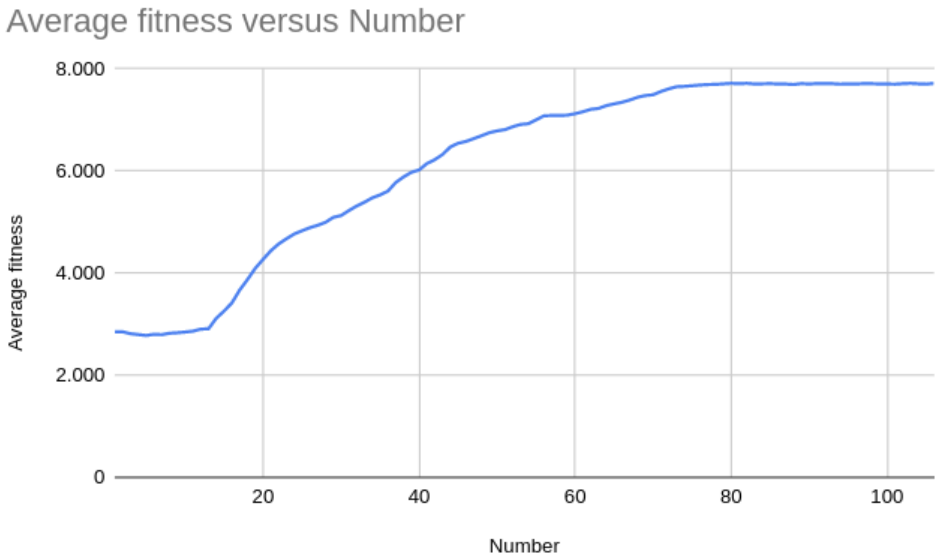

Figura 11 – melhor resposta da instância knapPI_1_100_1000_1

3.0.4.5 v1.5

Na versão 1.5 buscamos gerar uma população a partir de uma solução baseada em um algoritmo construtivo. Para gerar essa solução foi utilizado o valor relativo dos genes que podem ser colocados dentro de um cromossomo. Os genes foram ordenados em ordem decrescente em relação a esse valor, depois foram adicionados um a um respeitando a capacidade do cromossomo, para cada item adicionado o valor da capacidade disponível do cromossomo é atualizado. Quando o cromossomo não consegue adicionar mais itens tem se uma solução válida que pode ser melhorada por outro algoritmo. A partir dessa solução é gerado modificações nos genes criando assim novos indivíduos que formam a população inicial, a ideia é tentar criar uma população em que o algoritmo genético consiga mais facilmente chegar em uma resposta ótima. Tabela dos testes realizados:

Instância	pior resposta	melhor resposta	valor ótimo da instância
f1_l-d_kp_10_269	293	295	295
f2_l-d_kp_20_878	1018	1024	1024
f8_l-d_kp_2_10000	9734	9750	9767
knapPI_1_100_1000_1	7645	8810	9147

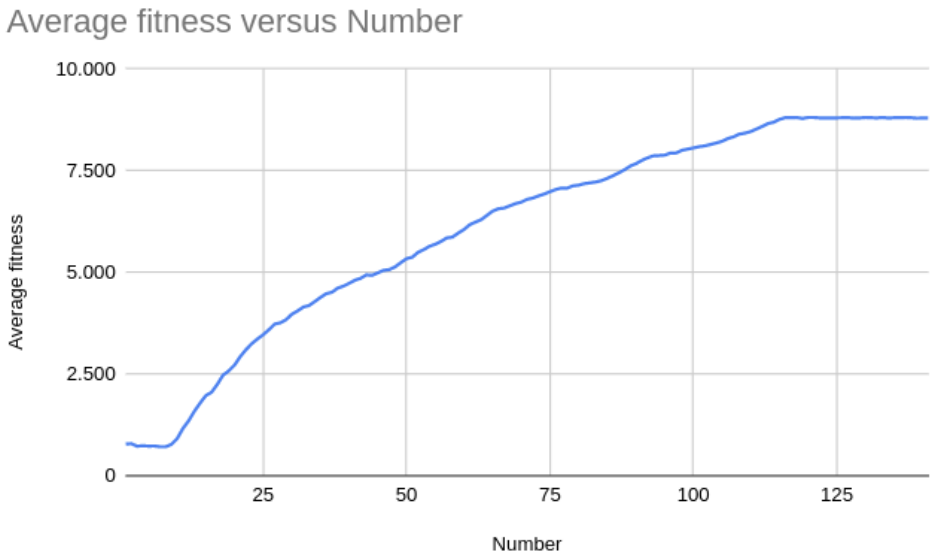

Figura 12 – melhor resposta da instância knapPI_1_100_1000_1

Houve uma melhoria na qualidade da solução, algumas ainda não continuam chegando perto de seu valor ótimo porém temos uma ligeira melhoria. O que foi percebido é que de alguma forma em algumas circunstâncias o algoritmo genética faz muitas iterações, mas do que na forma antiga de gerar a população inicial, e isso pode acusar que o programa demora mais tempo para chegar em uma população com pouca variação de valores da função *fitness*.

3.0.4.6 Visão geral das versões

Em resumo podemos ver cada versão da seguinte forma:

versão	comentário
v1.0	Primeira versão, tem a necessidade de configurar o número de iterações do algoritmo no código. Se o número não for o bastante pode ser que o programa não devolva uma resposta satisfatória.
v1.1	Mudança no tamanho da população limitando à 100 indivíduos
v1.3	É criado um meio de parada, uma forma de mutação dos genes dos filhos criados pela função de cruzamento. Por fim possui uma maneira de salvar os dados do programa.
v1.4	Altera a população que vai ser selecionada para o cruzamento, a fim de proporcionar maior variabilidade genética.
v1.5	Há uma modificação na forma de se obter a população inicial

4 COMPARAÇÃO COM A SOLUÇÃO UTILIZANDO ALGORITMOS DE CONSTRUÇÃO E MELHORAMENTO

Com os dados do primeiro trabalho da disciplina de Modelagem e Otimização Algorítmica é possível fazer um comparativo das implementações desenvolvidas.

A implementação do primeiro trabalho, que se utilizou de heurísticas de construção e melhoramento de uma maneira geral obteve respostas muito próximas da solução ótima, senão a própria, de uma forma bem direta. Já o algoritmo genético possui o fator da aleatoriedade que pode prejudicar o programa chegar perto da solução ótima. Um outro ponto é que processos que utilizam algoritmos genéticos tendem a perder a diversidade da população com o passar das interações, fazendo com que haja uma dificuldade em encontrar soluções ótimas, e com isso é necessário procurar por soluções que contornam esse problema.

Na opinião dos integrantes do grupo, o primeiro programa que se utiliza das heurísticas de construção e melhoramento possui um maior atrativo, sejam os conceitos que necessitamos para entendê-los, seja pelo seu desempenho chegando bem próximo da solução ótima, o que nem sempre acontece com o algoritmo genético desenvolvido para o segundo trabalho.

5 CONCLUSÃO

Como pode ser visto até aqui, o algoritmo genético simula algo que acontece na natureza. Essa metaheurística possui operações e um "fluxo" de operações que podem ser customizadas para o problema que se deseja resolver. Para o problema da mochila, o programa desenvolvido teve um desempenho muito bom para instâncias pequenas, porém isso não vale para instâncias maiores, onde se obteve respostas distantes do valor ótimo. Foi possível explorar o comportamento e a evolução de uma população com o passar das iterações do programa, o que nos revelou informações interessantes que ajudaram a criar inovações para serem implementadas. O seu desempenho talvez não seja tão bom quanto outras formas de resolver seu problema, mas a forma como essa metaheurística funciona é bem diferenciada.

REFERÊNCIAS

ALGORITMOS Genéticos. <http://www.nce.ufrj.br/GINAPE/VIDA/alggenet.htm#:~:text=Como%20crit%C3%A9rios%20de%20parada%20do,um%20determinado%20par%C3%A2metro%20do%20problema.> (Accessed on 11/03/2022).

DENG, L.; YANG, P.; LIU, W. An improved genetic algorithm. In: **2019 IEEE 5th International Conference on Computer and Communications (ICCC)**. [S.l.: s.n.], 2019. p. 47–51.

RECOMBINAçãO (computação evolutiva) – Wikipédia, a enciclopédia livre. <https://pt.wikipedia.org/wiki/Recombina%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o_evolutiva)>. (Accessed on 11/09/2022).

TALBI, E.-G. **Metaheuristics**. Hoboken, NJ: Wiley-Blackwell, 2009. (Wiley Series on Parallel and Distributed Computing).

WNG, J.; SUN, Q.; JIA, H. A novel method for solving knapsack problem base on hybrid genetic algorithm. In: **2010 2nd International Conference on Industrial and Information Systems**. [S.l.: s.n.], 2010. v. 1, p. 34–36.

ZHAO, J.; HUANG, T.; PANG, F.; LIU, Y. Genetic algorithm based on greedy strategy in the 0-1 knapsack problem. In: **2009 Third International Conference on Genetic and Evolutionary Computing**. [S.l.: s.n.], 2009. p. 105–107.