Universidade Tecnológica Federal do Paraná
Campus Apucarana
Bacharelado em Engenharia da Computação

Relatório Atividade Prática 3 - Snake

Alunos:
Ana Carolina Ribeiro Miranda
Eiti Parruca Adama
Lucas Eduardo Pires Parra

Novembro
2023

Universidade Tecnológica Federal do Paraná
Campus Apucarana
Bacharelado em Engenharia da Computação

Relatório Atividade Prática 3 - Snake

Terceiro Relatório da Atividade Prática apresentado na disciplina de Sistemas Inteligentes 1 do Curso de Engenharia de Computação da Universidade Tecnológica Federal do Parana, campus Apucarana como requisito parcial para aprovação da disciplina.

Aluno:
Ana Carolina Ribeiro Miranda
Eiti Parruca Adama
Lucas Eduardo Pires Parra

Novembro
2023

Conteúdo

1 Resumo

O presente trabalho representa a terceira atividade prática da disciplina e tem como propósito a concepção de um Algoritmo Genético (AG) capaz de controlar o jogo da cobrinha (*"Snake"*). Para esse fim, utiliza-se para facilitar o desenvolvimento, uma estrutura de jogo implementada com programação orientada a objetos (POO) em *Python*, usando a biblioteca *PyGame*. A plataforma utilizada para o desenvolvimento foi o *Visual Studio Code*.

O objetivo principal deste projeto é desenvolver um algoritmo genético capaz de gerar, avaliar e recombinar diferentes indivíduos de uma população, visando maximizar a pontuação, e por fim, selecionando o melhor ao final da execução.

2 Apresentação

O presente trabalho representa um marco significativo para o aprendizado e aplicação prática da inteligência artificial, com esta atividade sendo a terceira incursão na disciplina de sistemas inteligentes 1, onde realizou-se a implementação das funções necessárias para o funcionamento do AG, onde os indivíduos gerados são avaliados, recombinados,e os melhores são selecionados ao final da execução, com o objetivo de maximizar a pontuação e o tamanho da "*Snake*"tanto quanto possível, encontrando caminhos eficientes para coletar comida.

Utilizando a plataforma *Visual Studio Code* e fazendo uso da linguagem de programação *Python*, nossa missão envolveu a análise dos arquivos fornecidos, compreensão de seu conteúdo e, em seguida, a modificação e implementação do código necessário para o funcionamento do AG.

3 Descrição de atividades

Os principais objetos presentes no código desta atividade são:

- ***GeneticAlgorithm (GeneticAlgorithm.py)***: o *GeneticAlgorithm* é uma classe que define um Algoritmo Genético (AG) e seus operadores.

- ***Snake (Snake.py)***: outra classe de implementação do jogo "*Snake*" usando *PyGame*

- ***Main (main.py)***: arquivo principal que instancia e roda jogos de "*Snake*"

No contexto de Algoritmos Genéticos (AG), o termo "indivíduo" refere-se a uma solução candidata ou a uma representação potencial de uma solução para o problema em foco, no nosso caso onde o objetivo é controlar o jogo da cobrinha por meio de um AG, cada "indivíduo" no AG representa uma estratégia para as ações da cobrinha no jogo, sendo codificado como uma sequência de movimentos, como "ir para cima", "ir para baixo", "virar à esquerda" e "virar à direita". A avaliação desses indivíduos é realizada por meio de uma função de "*fitness*" para definir um parâmetro a ser avaliado a fim de se obter o melhor "indivíduo". A manipulação, avaliação, recombinação e seleção desses indivíduos ao longo das gerações constituem passos cruciais em um Algoritmo Genético, visando encontrar soluções cada vez mais eficazes para o problema em questão.

O AG gera uma população inicial, avalia o "*fitness*" de cada indivíduo, realiza operações de seleção, "*crossover*" e mutação ao longo de múltiplas gerações. Os resultados, como o melhor indivíduo, melhor "*fitness*", quantidade de alimentos coletados e o número total de gerações, são monitorados e exibidos durante a execução do algoritmo.

A função de "*fitness*" implementada na classe "*GeneticAlgorithm*", tem como função que recebe um indivíduo, representado por uma sequência de movimentos, e executa o jogo da cobrinha com base nesses movimentos, retornando uma pontuação de "*score*" que será usada para o calculo do "*fitness*", conforme o pseudocódigo abaixo:

Figura 1: Pseudocódigo da função fitness

```
função fitnees():

    // criar o jogo da snake com uma seed aleatória
    jogo = SnakeGame(semente)

    // atribuir os movimentos do indivíduo ao jogo
    jogo.movimentos = indivíduo

    // executar o jogo e obter a pontuação e os alimentos consumidos
    pontuação, alimentos = jogo.executar()

    // adicionar incentivos à pontuação
    // o comprimento da cobra é multiplicado por 10
    aptidão_comprimento = tamanho(jogo.corpo_cobra) * 10
    // o tempo de sobrevivência é multiplicado por 100
    aptidão_tempo = (jogo.tempoFinal - jogo.tempoInicial) * 100

    // verificar se a distância da fruta diminuiu ou aumentou e atualizar a pontuação
    se diminuiu:
        pontuação = pontuação + 50
    senão:
        pontuação = pontuação - 50

    // função de aptidão
    pontuação = pontuação + aptidão_comprimento + aptidão_tempo

    // impedir pontuação negativa
    se pontuação <= 0:
        pontuação = 1

    // retornar a pontuação e os alimentos
    retornar pontuação, alimentos
```

Fonte: Autoria própria.

A pontuação "*score*"e a quantidade de comidas coletadas "*foods*"são obtidas a partir do jogo, refletindo o desempenho do indivíduo, onde a pontuação final de um indivíduo recebe bonificações ou penalidades levando em conta a quantidade de comida, o tempo sobrevivido e também se diminuiu ou não a sua distancia em relação a fruta. Caso a pontuação fique negativa, é alterada para 1 a fim de evitar resultados indesejados.

A proposta abordada optou pela realização da seleção de indivíduos por meio do método de torneio, onde o tamanho é definido pela variável "$tournamentSize = 3$"na classe "*GeneticAlgorithm*"que determina quantos indivíduos são escolhidos aleatoriamente para participar do torneio, o método do torneio é implementado na função "*selectParents*"onde são realizados torneios para selecionar os dois melhores indivíduos de cada torneio como pais. A implementação da função foi realizada seguindo a estrutura do pseudocódigo abaixo:

Figura 2: Pseudocódigo da função de seleção

```
função seleção():

    // selecionar aleatoriamente indivíduos para o torneio
    candidatos <- escolhas_aleatórias(população, k = tamanho_torneio)

    // avaliar o desempenho de aptidão(fitness) dos candidatos
    valores_aptidão <- [aptidão[índice(população, candidato)] para candidato em candidatos]

    // obter os índices dos dois candidatos com o melhor desempenho de aptidão
    índices_vencedores <- ordenar(índices(valores_aptidão))[:2]

    // retornar os dois vencedores do torneio
    pais <- [candidatos[i] para i em índices_vencedores]
    retornar pais
```

Fonte: Autoria própria.

O "*crossover*", uma espécie de reprodução dos indivíduos, é realizada na função "*generateChildren*", da classe "*GeneticAlgorithm*", onde recebe dois pais e gera dois filhos utilizando o *crossover* de um ponto, uma técnica de recombinação genética que consiste em escolher um ponto aleatório de corte e trocar as partes finais de dois indivíduos para gerar novos descendentes. Essa técnica visa explorar a diversidade genética e criar soluções melhores. A função implementada seguiu a estrutura do pseudocódigo abaixo.

Figura 3: Pseudocódigo da função de crossover

```
função crossover():

    // garante que o ponto de corte não é o início ou o final do cromossomo
    ponto_corte <- número_aleatório(1, tamanho(pai) - 1)

    // realiza o crossover
    filho1 <- pai1[:ponto_corte] + pai2[ponto_corte:]
    filho2 <- pai2[:ponto_corte] + pai1[ponto_corte:]

    // retorna os dois filhos
    retornar filho1, filho2
```

Fonte: Autoria própria.

Já a mutação é realizada na função "*mutationOperator*"na classe "*GeneticAlgorithm*", essa função percorre cada movimento de um indivíduo e, com uma probabilidade determinada pela taxa de mutação chamada "*mutationRate*", realiza uma mutação, ou seja, troca o gene, o movimento, por um novo movimento aleatório, assim cada gene tem uma probabilidade de ser mutado, introduzindo diversidade na população e explorando novas regiões

do espaço de busca.

Figura 4: Pseudocódigo da função de mutação

```
função mutação():

    // percorre cada gene do filho
    Para i de 0 até tamanho(filho) - 1 faça

        // gera um número aleatório entre 0 e 1
        Se número_aleatório() < taxa_mutação então

            // escolhe um movimento aleatório entre 'UP', 'DOWN', 'LEFT' e 'RIGHT'
            filho[i] <- escolha_aleatória(['UP', 'DOWN', 'LEFT', 'RIGHT'])

    // retorna o filho mutado
    retornar filho
```

Fonte: Autoria própria.

Todas essas classes representam partes do algoritmo genético elaborado, que são juntadas na função "*execute*", da classe já mencionada anteriormente, e então a lógica de funcionamento é aplicada, onde, inicia-se gerando uma população inicial, que em seguida tem seu "*fitness*"avaliado, realiza-se então a seleção por torneio, que retorna dois "pais", que então geram dois "filhos"que podem ou não sofrer uma mutação, com isso se repetindo até que uma nova população seja formada, pela quantidade de gerações estipuladas.

Já a classe "*SnakeGame*", é responsável por gerenciar todos os aspectos do jogo, incluindo a movimentação da cobra, a geração de frutas, o controle de pontuação e a lógica do jogo, a cobra é representada por uma série de segmentos que compõem seu corpo. A inicialização do jogo envolve a definição de atributos essenciais, como a velocidade da cobra (*snake_speed*), as dimensões da janela do jogo (*window_x* e *window_y*), a posição inicial da cobra (*snake_position*), a direção inicial (*direction*).

O método *run* implementa o ciclo principal do jogo, gerenciando a lógica de movimentação da cobra, a detecção de colisões, a atualização da pontuação e a exibição visual em tempo real. A lógica de colisão verifica se a cobra atinge as bordas da janela ou seu próprio corpo, decrementando a pontuação, já se ela coleta uma fruta ou fica mais próximo dela, a pontuação é incrementada, proporcionando uma dinâmica de jogo que recompensa a eficiência na coleta de alimentos.

Outra modificação realizada é da classe *Colors.py* ao qual foi adicionado uma função chamada "corzinha"que permite gerar uma cor aleatória, sendo utilizada na classe "*SnakeGame*"para modificação da cor da cobrinha a cada movimento.

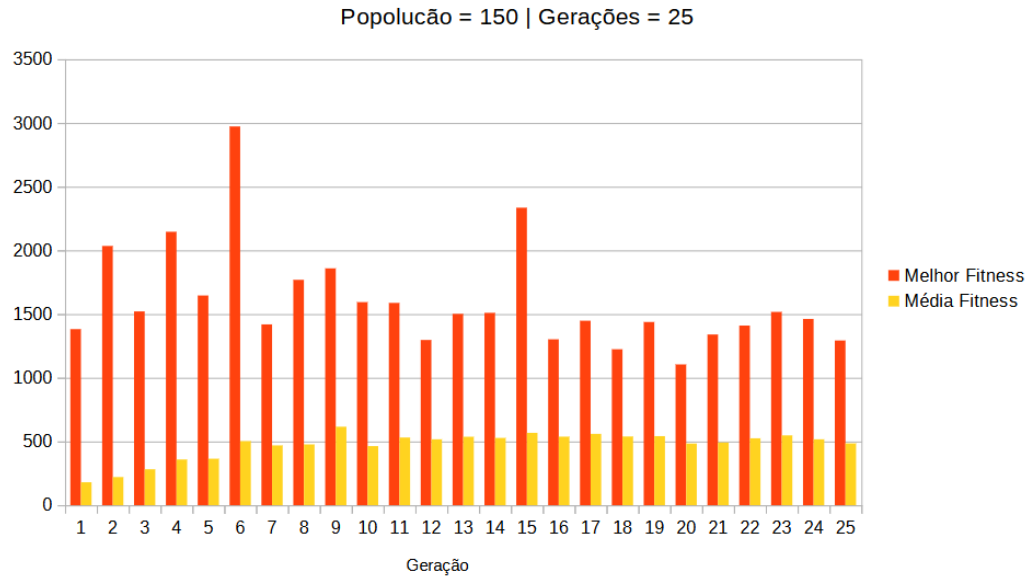
Por fim a classe "*Main*", onde são importados os módulos essenciais mencionados anteriormente, temos a váriavel "*seed*"responsável por um valor utilizado para iniciar o gerador de números aleatórios, a fim de garantir a reprodutibilidade dos resultados, e o "*GeneticAlgorithm*"seguida pela chamada do método "*execute*"para evoluir a população de agentes e obter métricas para obter o melhor indivíduo, mostrando o número numero de gerações, o total de comidas consumidas por geração, média de "*fitness*"por geração e o melhor "*fitness*"por geração.

4 Análise dos Resultados

A definição dos operadores genéticos, como *crossover* e mutação, é diretamente afetada pela forma como os indivíduos são codificados, e a escolha da codificação influencia como as informações genéticas são representadas e manipuladas durante o processo evolutivo, como dito no tópico anterior, os indivíduos são codificados como sequências de movimentos possíveis para a cobrinha no jogo da *Snake*, cada movimento é representado por uma *string* que pode ser *"UP', "DOWN', "LEFT' ou "RIGHT'*, essa escolha de codificação é devido os genes (movimentos) serem discretos e representam as decisões do agente durante o jogo. Cada indivíduo gerado recebe uma sequencia de movimentos, que variam de forma aleatória em quantidade, entre 10 e 500. Porém, visando tornar mais fácil a analise do resultado gerado pelo algoritmo através da reprodutibilidade dos resultados, fixou-se a seed da sequência de valores aleatórios em 33 e a taxa de mutação em 5%.

Realizou-se então a execução do algoritmo para diferentes tamanhos de população e gerações, primeiramente para uma população de 150 indivíduos, por 25 gerações, com os resultados mostrados na figura 5 abaixo, que apresenta o melhor "*fitness*"por geração e a média de "*fitness*":

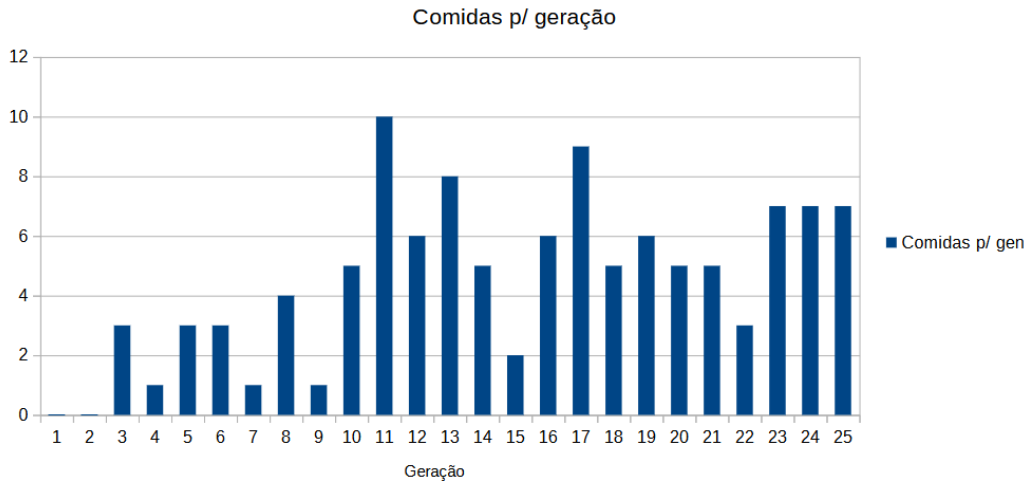
Figura 5: Média fitness e melhor p/ geração



Fonte: Autoria própria

Criou-se também o gráfico da figura 6, mostrando a quantidade de comidas por geração:

Figura 6: Comidas p/ geração

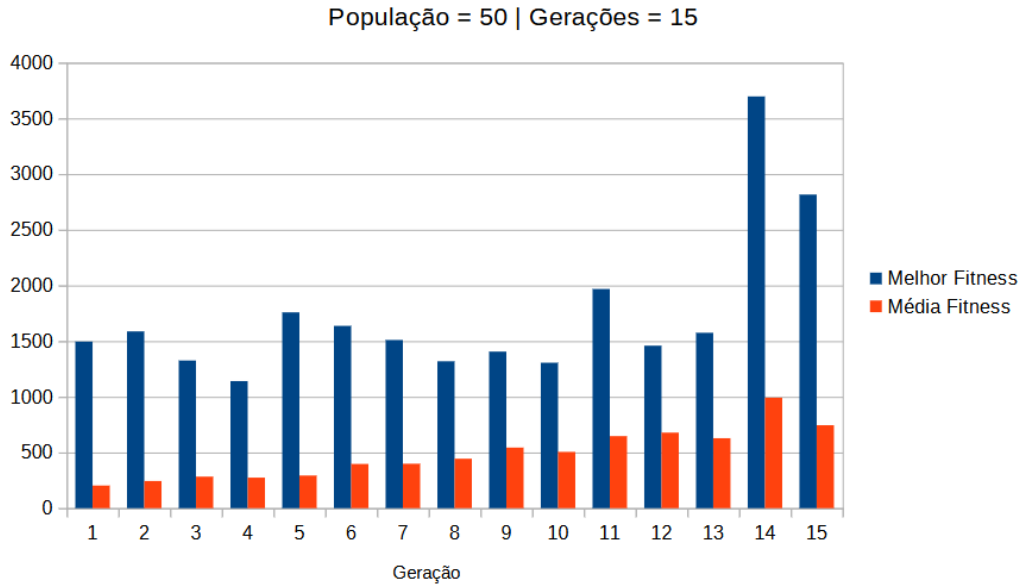


Fonte: Autoria própria

Observa-se no gráfico da figura 5 uma maior oscilação no valor do melhor "*fitness*", se comparado com o comportamento da média do "*fitness*", que após cerca de 10 gerações, manteve-se em torno de 600 pontos. Já na figura 6, nota-se uma tendencia a priorizar uma maior quantidade de comidas ao passar das gerações.

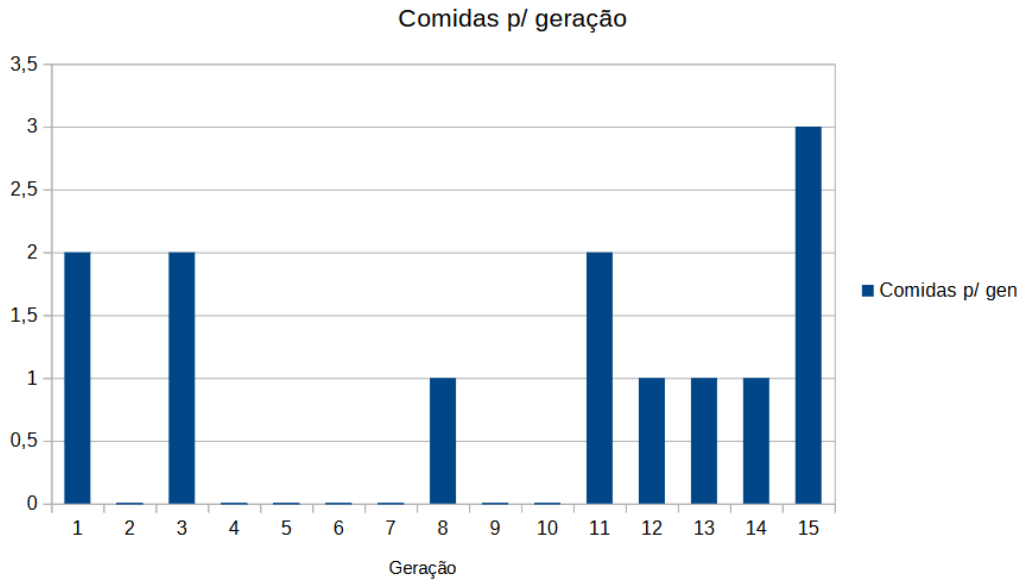
Realizando a execução do algoritmo para uma população de 50 indivíduos, por 15 gerações, obteve-se os dados mostrados nos gráficos das figuras 7 e 8, logo abaixo:

Figura 7: Média fitness e melhor p/ geração



Fonte: Autoria própria

Figura 8: Comidas p/ geração

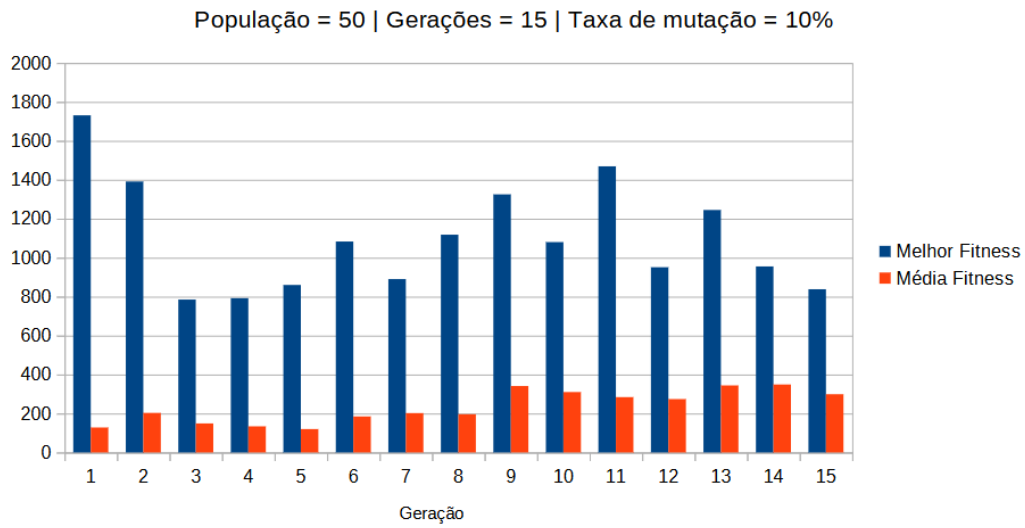


Fonte: Autoria própria

Utilizando esse mesmo tamanho de população e quantidade de gerações,

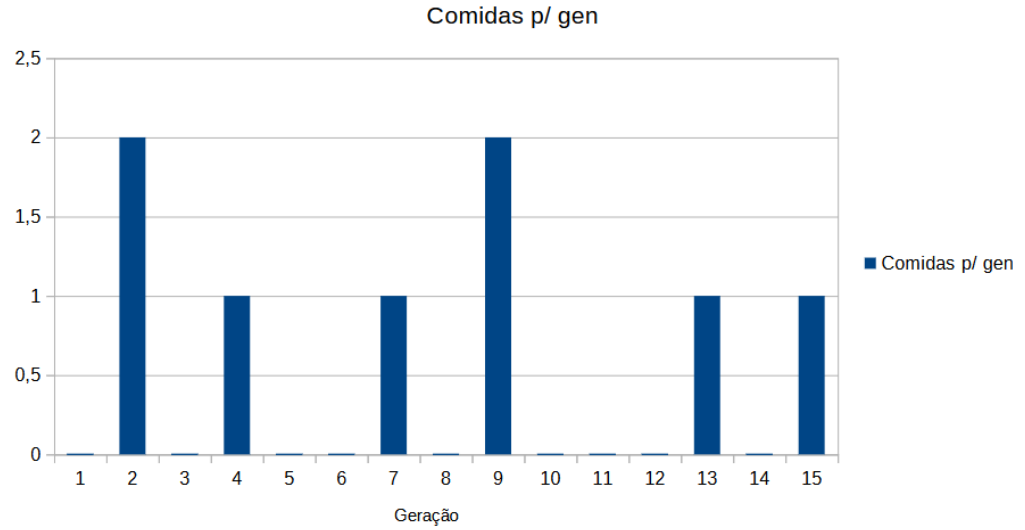
porém alterando-se a taxa de mutação de 5% para 10%, obteve-se os dados mostrados nos gráficos das figuras 9 e 10:

Figura 9: Média fitness e melhor p/ geração - 10%



Fonte: Autoria própria

Figura 10: Comidas p/ geração - 10%

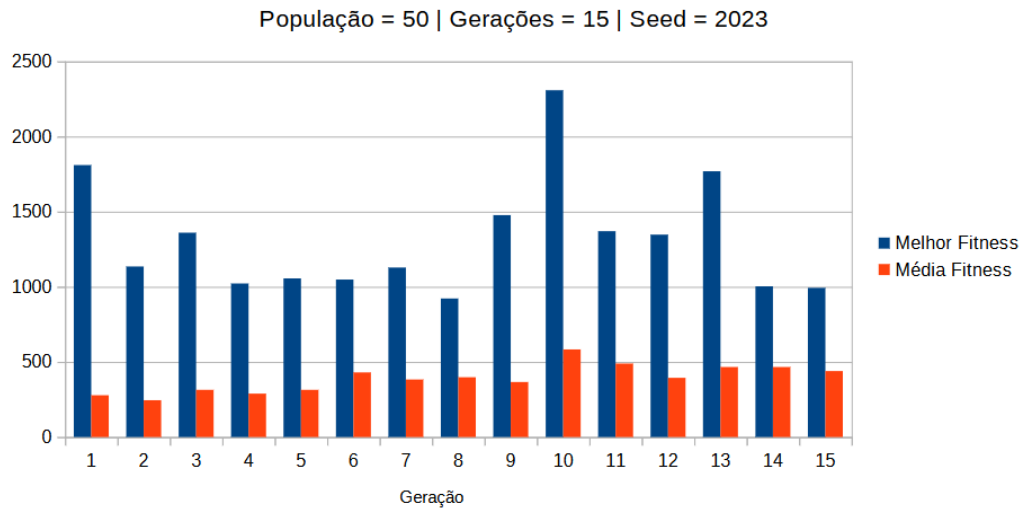


Fonte: Autoria própria

Utilizando esse mesmo tamanho de população e quantidade de gerações,

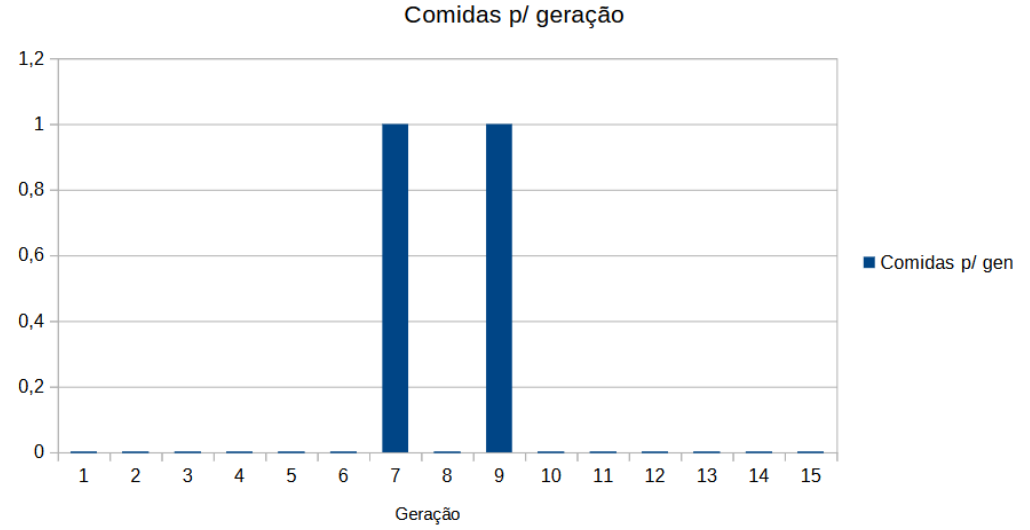
voltando a taxa de mutação para 5%, porém alterando a "*seed*"de 33 para 2023, obteve-se os dados mostrados nos gráficos das figuras 11 e 12:

Figura 11: Média fitness e melhor p/ geração - Seed = 2023



Fonte: Autoria própria

Figura 12: Comidas p/ geração - Seed = 2023



Fonte: Autoria própria

Comparando esses 3 casos, percebe-se que os melhores "*fitness*"e média

dos "*fitness*"das figuras 7 e 8 apresentam valores melhores se comparados aos das figuras 9, 10, 11 e 12, apesar de todos manterem o comportamento oscilatório do melhor "*fitness*"e o comportamento de subida e estabilização da média dos "*fitness*". Nota-se támbem que a mudança na taxa de mutação e na "*seed*"afetaram o busca por comidas, havendo uma quebra no comportamento de buscar aumentar a quantidade de comidas por geração.

Os resultados obtidos se mostraram aceitáveis, com o AG sendo capaz de gerar movimentos inteligentes para a cobra, que conseguiu comer várias frutas e aumentar o seu comprimento. No entanto, também foram observados alguns problemas, como a cobra ficar presa em "*loops*"ou se chocar com o seu próprio corpo. Além disso, o AG não foi capaz de explorar todo o espaço de busca, pois a representação usada limita o número de movimentos possíveis.

5 Trabalhos Futuros

Como trabalho futuro, sugere-se melhorar a representação dos indivíduos, usando uma codificação mais flexível e adaptativa, que permita variar o número e o tipo de movimentos. Também sugere-se testar outros métodos de seleção, cruzamento e mutação, que possam aumentar a diversidade genética da população e evitar a convergência prematura.

6 Referências

1. LUGER, George F. Inteligência artificial. 6. ed. São Paulo, SP: Pearson Education do Brasil, 2013. xvii, 614 p. ISBN 9788581435503.

2. RUSSELL, Stuart J.; NORVIG, Peter. Inteligência artificial. Rio de Janeiro, RJ: Elsevier,2013. 988 p. ISBN 9788535237016.

3. LUKE, Sean. Essentials of Metaheuristics. Second edition. Lulu: 2013. Available for freeat `http://cs.gmu.edu/$\sim$sean/book/metaheuristics/`

4. DE JONG, Kenneth A. Evoluationary Computation: A Unified Approach. The MIT Press: 2006.