Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Apucarana
Bacharelado em Engenharia de Computação

Universidade Tecnológica Federal do Paraná

Engenharia de Computação

ALGORITMOS GENÉTICOS APLICADO A RESOLUÇÃO DO JOGO SNAKE

Aluno: Gabriel Finger Conte

João Vitor Garcia Carvalho

Maria Eduarda Pedroso

Apucarana

Novembro/2023

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Apucarana
Bacharelado em Engenharia de Computação

Universidade Tecnológica Federal do Paraná

Engenharia de Computação

ALGORITMOS GENÉTICOS APLICADO A RESOLUÇÃO DO JOGO SNAKE

Relatório da Atividade 03, apresentado à disciplina de Sistemas Inteligentes 01 do Curso Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção de nota no semestre.

Aluno: Gabriel Finger Conte

João Vitor Garcia Carvalho

Maria Eduarda Pedroso

Apucarana

Novembro/2023

SUMÁRIO

1. RESUMO

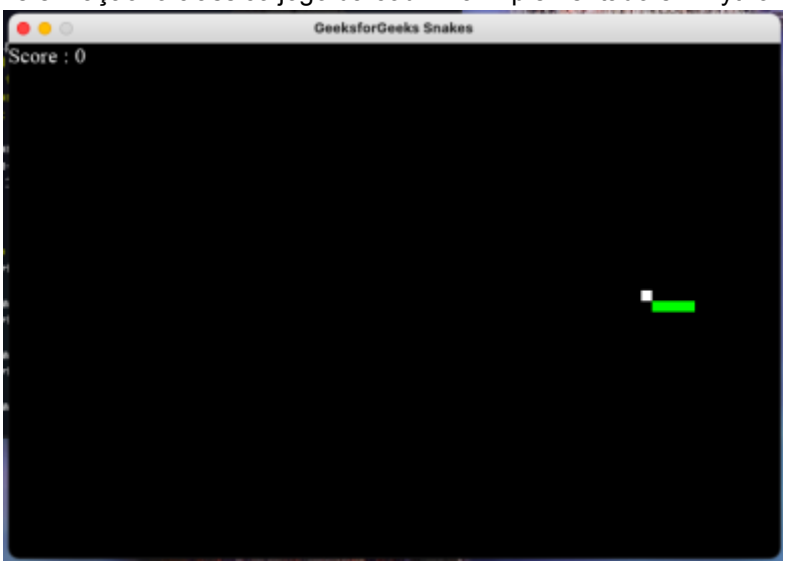
Este relatório documenta a implementação de um Algoritmo Genético (AG) para controlar o jogo Snake em Python com a biblioteca Pygame. A estrutura do projeto envolveu a criação de uma classe GeneticAlgorithm para gerenciar o AG, incluindo métodos para geração de população, avaliação de fitness, seleção, crossover, mutação e execução ao longo de várias gerações. A lógica do jogo Snake foi encapsulada na classe SnakeGame, que utiliza o Pygame para manipular a interface gráfica e a mecânica do jogo. A integração entre o AG e o Snake permitiu que a IA controlasse a cobra, adaptando métodos para avaliar a aptidão com base no desempenho no jogo. O relatório enfatiza a importância de cada componente, encorajando experimentações com configurações do AG, operadores genéticos e parâmetros do Snake. A sugestão de execução de testes com diferentes sementes visa analisar o comportamento estocástico do AG. O projeto ofereceu uma valiosa oportunidade de compreender e implementar Algoritmos Genéticos em um cenário de jogo desafiador, detalhando escolhas de implementação, procedimentos de teste e conclusões. Contribui para uma compreensão mais profunda da aplicação de AG em ambientes de jogos.

2. INTRODUÇÃO

Problemas relacionados à busca por estratégias eficientes, seja para otimização de trajetos ou atendimento a critérios específicos, têm sido um desafio constante no campo da ciência da computação. Atualmente, esses desafios são abordados e refinados através do desenvolvimento contínuo de algoritmos. Essas soluções encontram aplicação em diversas áreas, desde dispositivos de GPS para navegação até a concepção de estratégias em videogames.

Com base nesse contexto, o presente trabalho se propõe a aplicar conhecimentos adquiridos na disciplina de Sistemas Inteligentes 1 para desenvolver e aprimorar um algoritmo genético aplicado ao jogo Snake. O jogo foi implementado em Python, utilizando a biblioteca Pygame, e a metodologia adotada envolve a modificação de quatro arquivos principais: main.py, SnakeGame.py, Colors.py e GeneticAlgorithm.py.

Figura 1 - Snake em ação: o clássico jogo da cobrinha implementado em Python, via PyGame.



Fonte: [2]

O arquivo main.py desempenha um papel central, funcionando como ponto de entrada e execução do programa. Sua lógica é simplificada, importando a lógica do jogo Snake e a biblioteca random. Uma seed fixa é definida para garantir a reprodutibilidade do algoritmo genético. A função principal cria uma instância do jogo

Snake e o coloca em execução, utilizando bibliotecas como random, matplotlib, copy, pygame e time. A execução do código é realizada através do terminal de comando, iniciando o interpretador Python.

Já o arquivo SnakeGame.py abriga a lógica de execução do jogo Snake e foi modificado para incorporar o algoritmo genético. A classe SnakeGame é definida com atributos necessários para a emulação do jogo, incluindo dois novos atributos: vetor_direcoes e genetic_algorithm. O vetor_direcoes armazena a solução encontrada pelo algoritmo genético. A função para rodar o jogo é elaborada dentro de um loop infinito while(true), informando a posição da cobra e da fruta, bem como a direção atual da cobra para o método execute da instância genetic_algorithm. A solução é então salva no vetor_direcoes, e o jogo é atualizado de acordo com as direções encontradas.

O arquivo GeneticAlgorithm.py, por sua vez, representa o núcleo do algoritmo genético desenvolvido para encontrar um conjunto de direções que maximize o tempo de vida da cobra no jogo Snake. A representação dos indivíduos é feita por vetores de direções, considerando as quatro direções cardeais (Cima, Baixo, Esquerda e Direita). A população inicial é gerada aleatoriamente, e a avaliação do fitness de cada indivíduo é realizada com base na simulação do jogo, considerando critérios como tempo de sobrevivência, consumo de frutas e distância até a fruta.

O método de seleção dos pais adota um torneio simples de três indivíduos, e o crossover é implementado em três variantes: Crossover de um ponto, Crossover de dois pontos e Crossover uniforme. Quanto à mutação, são empregados três métodos distintos: Inversão, Deslocamento e Mutação uniforme. Um critério de pânico é introduzido, aumentando a taxa de mutação em situações específicas.

A seção de Resultados e Discussão apresenta a funcionalidade do algoritmo desenvolvido, demonstrando resultados satisfatórios na maximização do desempenho da cobra no jogo Snake. Os gráficos exibem a convergência do algoritmo ao longo das gerações, destacando a importância do mecanismo de pânico para evitar máximos locais.

Ressalta-se que os métodos de crossover e seleção, juntamente com a introdução do elitismo, foram decisivos para o sucesso do algoritmo. A análise dos resultados evidencia a eficácia da abordagem adotada, com o algoritmo atingindo resultados satisfatórios em termos de desempenho da cobra no jogo Snake.

3. METODOLOGIA

O código do jogo Snake implementado em Python via biblioteca Pygame, disponibilizado pelo professor da disciplina no repositório [3], consiste em 4 arquivos fundamentais: main.py, SnakeGame.py, Colors.py e GeneticAlgorithm.py.

Salvo o terceiro arquivo, que contém o padrão de cores do jogo, foram modificados os demais arquivos de modo a implementar um algoritmo genético capaz de jogar o Snake, maximizando o tempo de vida da cobra.

A estrutura lógica e metodologia pensada para o projeto é melhor compreendida explorando detalhadamente a estrutura de cada um dos códigos adaptados. Para tanto, abordaremos em mais detalhes nas seções seguintes.

3.1. Arquivo Main.py

Tal arquivo condensa os demais códigos desenvolvidos, importando-os e servindo como interface para a execução do programa, conforme solicitado por Mantovani [2]. Sua lógica pode ser simplificada no seguinte pseudocódigo:

```
Python
import a lõgica do arquivo SnakeGame
import a biblioteca random

Defini-se uma seed fixa para a biblioteca random gerar os mesmo valores em
todas as execuções
```

```
def Função Main:
	Cria uma instância do jogo Snake
	Coloca o jogo para rodar
```

Assim, com as bibliotecas random, matplotlib, copy, pygame e time instaladas no ambiente de execução python, basta executar o terminal de comando na pasta com os arquivos e chamar o interpretador python para executar o código do arquivo através do comando “pyton main.py”.

3.2. Arquivo SnakeGame.py

Nesse arquivo, encontra-se a lógica de execução do jogo Snake, controlando a exibição e simulação do mesmo. Para a prática atual, realizaram-se algumas alterações pontuais na estrutura do programa de modo a implementar o algoritmo genético desenvolvido no controle da cobra. Resultando na lógica descrita pelo pseudocódigo a seguir.

```
Python
import das bibliotecas necessárias

Definição da class SnakeGame que representará o jogo:
	Definição dos atributos da classe, necessárias para emular o jogo
	# Adição de dois novos atributos:
	vetor_direcoes # Armazenará a solução encontrada pelo AG
	genetic_algorithm # Instância do AG desenvolvido para encontrar as soluções

	def Função de inicialização do jogo # Inalterada

	def Função para mostrar o placar # Inalterada
```

```
def Função para tratamento do fim de jogo # Inalterada

def Função para rodar o jogo:
      # Dentro do loop infinito while(true)
      Informa a posição da cobra e da fruta e a atual direção da cobra
      para o método execute da instância genetic_algorithm, de modo a
      obter a solução até a fruta atual
      Salva a solução encontrada no vetor_direcoes

      for Para cada direção no vetor de direções:
          Interpreta como se tivesse apertado uma das setas,
          dependendo do valor na posição (Cima, Baixo, Esquerda ou
          Direita)

          Atualização da posição da cobra # Inalterado

          Verificação se a cobra comeu uma fruta # Inalterado

          Se a fruta foi consumida, gera a próxima fruta
              # Região com a nova lógica
              Atualização da interface onde o jogo é mostrado #
              Inalterado
              Da um break, saindo do for para o vetor de direções e
              reaplicando o GA para encontrar uma solução até a nova
              fruta.

          Se não consumiu a fruta:
              Verificação das condiçoes de game-over # Inalterado
              Atualização da interface onde o jogo é mostrado #
              Inalterado
              Repete-se para a próxima direção do vetor solução
```

3.3. Arquivo GeneticAlgorithm.py

Arquivo principal contendo o algoritmo genético desenvolvido para achar um conjunto de direções para a cobra seguir, de modo a alcançar a fruta, seu objetivo, sem morrer. Ou ao menos chegar próximo a ela.

Apesar de ter sido utilizada a estrutura básica de um AG para a implementação atual, primeiramente faz-se necessário explanar a representação utilizada para as soluções, vulgo indivíduos de cada população. Como o Snake trata de um jogo onde controlamos a cobra pelo mapa, mudando a sua direção, optou-se por representar cada indivíduo como um vetor de direções, representando as quatro direções cardeais: Cima, baixo, esquerda e direita.

A fim de gerar a população inicial, definiu-se uma função que gera um vetor de indivíduos, pegando uma combinação aleatória das quatro direções cardeais.

Já para avaliar o fitness de cada indivíduo, separou-se a lógica em duas funções:

```
Python
def Função de fitness
    Recebe a posição e o corpo da cobra, bem como a posição da fruta, bem
    como a última direção que a mesma estava indo. Além do índivíduo a ser
    avaliado.
    Simula o jogo a partir da posição atual da cobra
        A cada movimento que a cobra está viva, adiciona-se uma unidade no
        tempo de sobrevivência.
        A cada movimento, verifica a distancia até a fruta. Onde se a mesma
        aumentou:
            aumenta a variável de registro da distancia em uma unidade.
        Se não:
            reduz a variável de registro da distancia em uma unidade.
        Verifica se a cobra não pegou uma fruta, em caso afirmativo indica
        que a mesma pegou a fruta e sai da simulação.
```

```
        Verifica se a cobra não morreu, batendo na parede ou em si mesma, em
        caso afirmativo finalizando a simulação e indicando que a mesma
        morreu.
    Após o fim da simulação, return se uma tupla informando se a mesma
    permaneceu viva, o tempo de sobrevivência, se a mesma pegou a fruta, se a
    mesma morreu.

def Função de avaliação do indivíduo
    Chama a função de fitness para pegar as métricas
    Calcula um valor base para o fitness como 1000 + tempo de sobrevivência #
    1000 para evitar um fitness negativo, que estava dando problema
    Determina se ganhará um bonus de 800 por sobreviver ou uma penalidade de
    -800 por morrer
    Determina se ganhará um bonus de 1000 por pegar a fruta ou uma penalidade
    de -300 por não pegá-la
    Calcula o fitness como a soma do valor base + bonus de sobrevivência +
    bonus por pegar a fruta + registro das distâncias

    Verifica se a elite com os maiores fitness está cheia, se não apenas
    coloca o indivíduo nela.
    Caso contrário, verifica se o fitness é maior que os já gerados,
    substituindo o membro de menor fitness dos membros da fitElite.

    Se o indivíduo pegou a fruta, verifica se já está na elite dos comedores
    de frutas, foodElite. Se não, adiciona ele no lugar do membro com o maior
    fitness, visto que indicaria que pegou a fruta em um tempo mais curto.

    return o fitness
```

O método de seleção dos pais utilizado foi um torneio simples de 3 indivíduos para o com maior fitness, mas deixou-se disponível a implementação do método da roleta desenvolvido para testes.

Para o crossover implementou-se 3 métodos distintos que ficaram disponíveis para seleção:

- Crossover de um ponto: selecionando um ponto aleatório de corte, gera os indivíduos filhos mesclando os dados antes e depois do ponto de corte dos indivíduos pais.
- Crossover de dois pontos: selecionando dois pontos aleatórios de corte, gera os indivíduos filhos mesclando os dados antes e depois com os dados entre os pontos de cortes dos indivíduos pais. Sendo o método preferido, deixado como padrão.
- Crossover uniforme: para cada posição, aleatoriamente seleciona aleatoriamente o valor de um dos pais para cada um dos filhos.

Como operador de mutação, para gerar uma maior variabilidade e possibilidade de mutar indivíduos para obter uma melhor solução, implementaram-se 3 métodos distintos. Sendo o método de mutação empregado determinado de maneira aleatória para cada vez que o operador é chamado.

- Inversão: selecionando aleatoriamente um segmento do indivíduo, inverte-se os valores de posição. Os últimos se tornam os primeiro, e vice versa.
- Deslocamento: selecionando aleatoriamente um segmento do indivíduo, desloca-se o mesmo para o final do mesmo.
- Uniforme: para cada posição do indivíduo, adiciona-se uma chance aleatória de mutação, que caso seja menor que o critério, seleciona aleatoriamente uma nova direção para a cobra seguir.

Além disso, adicionou-se um fator a mais no processo de mutação, chamado critério de pânico. Caso a população atual esteja em pânico - como será explicado logo em seguida - considera-se uma nova taxa de mutação mais elevada que o padrão. Tal processo é feito inspirado na biologia, onde caso uma população se encontre ameaçada, “em pânico”, os indivíduos irão deixar de se comportar de forma tão coesa e passar a ter um fator maior aleatoriedade. O que foi traduzido aumentando a taxa de mutação com esse critério de pânico.

Deixou-se disponível também uma função para avaliar o comportamento do melhor fitness e a média do fitness de cada geração, simplesmente plotando os valores com a biblioteca matplotlib.

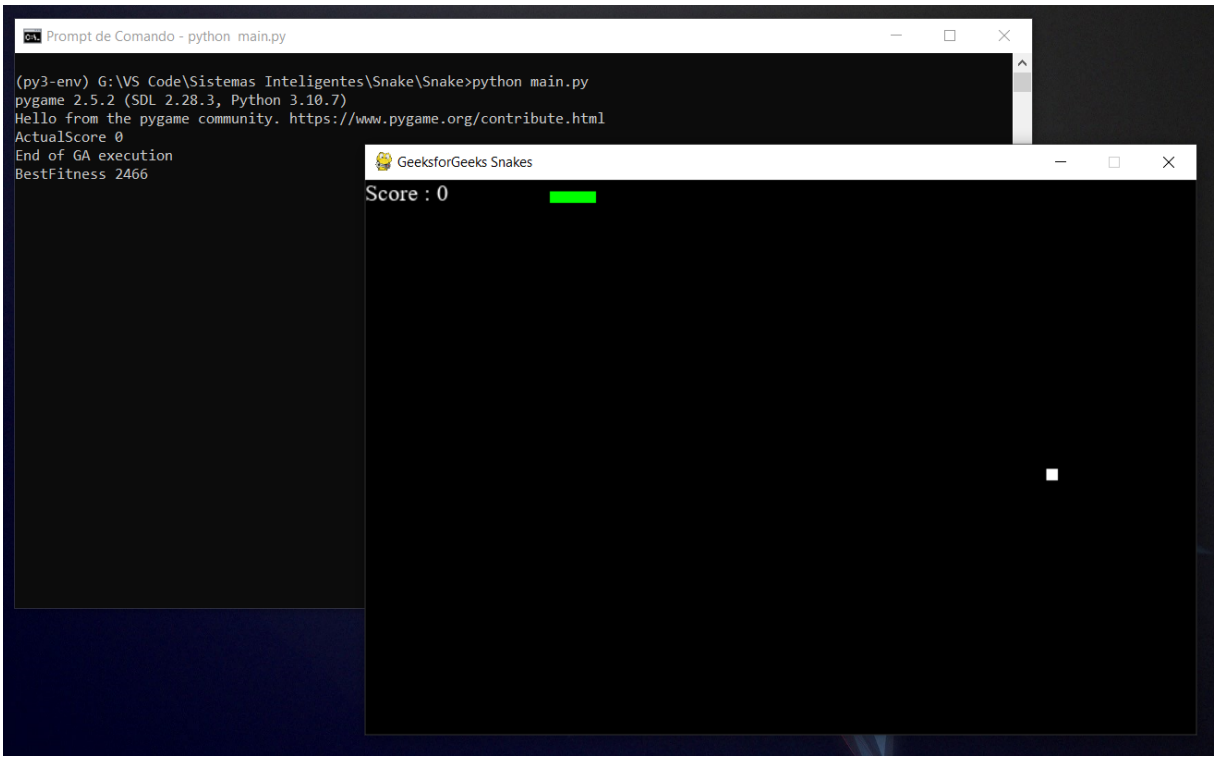
Por fim, chegamos na execução do algoritmo em si, a qual segue a lógica descrita no pseudo algoritmo abaixo:

```
Python
def Função de execução

        Inicia os vetores que armazenaram os dados para evitar sobrescrita
        Gera a população inicial
        Para cada geração

        Avalia o fitness dos indivíduos da população
        Determina e salva o melhor fitness
        Determina e salva a média de fitness da população

        Verifica se houve uma melhora no melhor fitness comparado a geração
        anterior. Se não houve melhora, aumenta o critério de pânico da
        população, se não reseta o nível de pânico.

        Seleciona-se os casais dentro da população
        Gera-se os filhos desses casais
        Aplicase o operador de mutação em cada filho

        Mantém as elites e substitui o restante da população pelos novos
        indivíduos
```

4. RESULTADOS E DISCUSSÃO

Após os testes e desenvolvimento do código, culminou-se na lógica descrita na metodologia. A partir deste algoritmo, como pode ser observado na Figura C, verificou-se a funcionalidade do algoritmo desenvolvido.

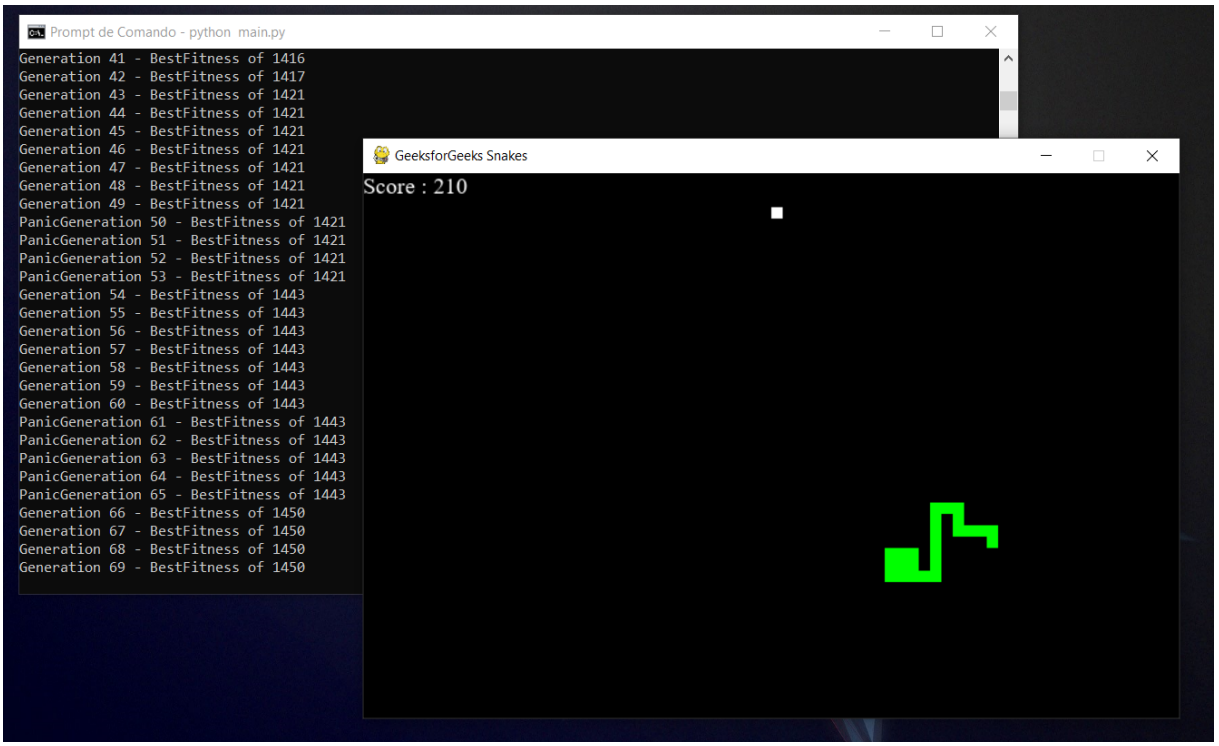
Figura C - Ilustração da inicialização do Snake com o AG desenvolvido



Fonte: autoria própria

Observou-se, como pode-se notar na Figura D, que o algoritmo desenvolvido é funcional, conseguindo manter a cobra viva de modo a conseguir um score relevante sem morrer.

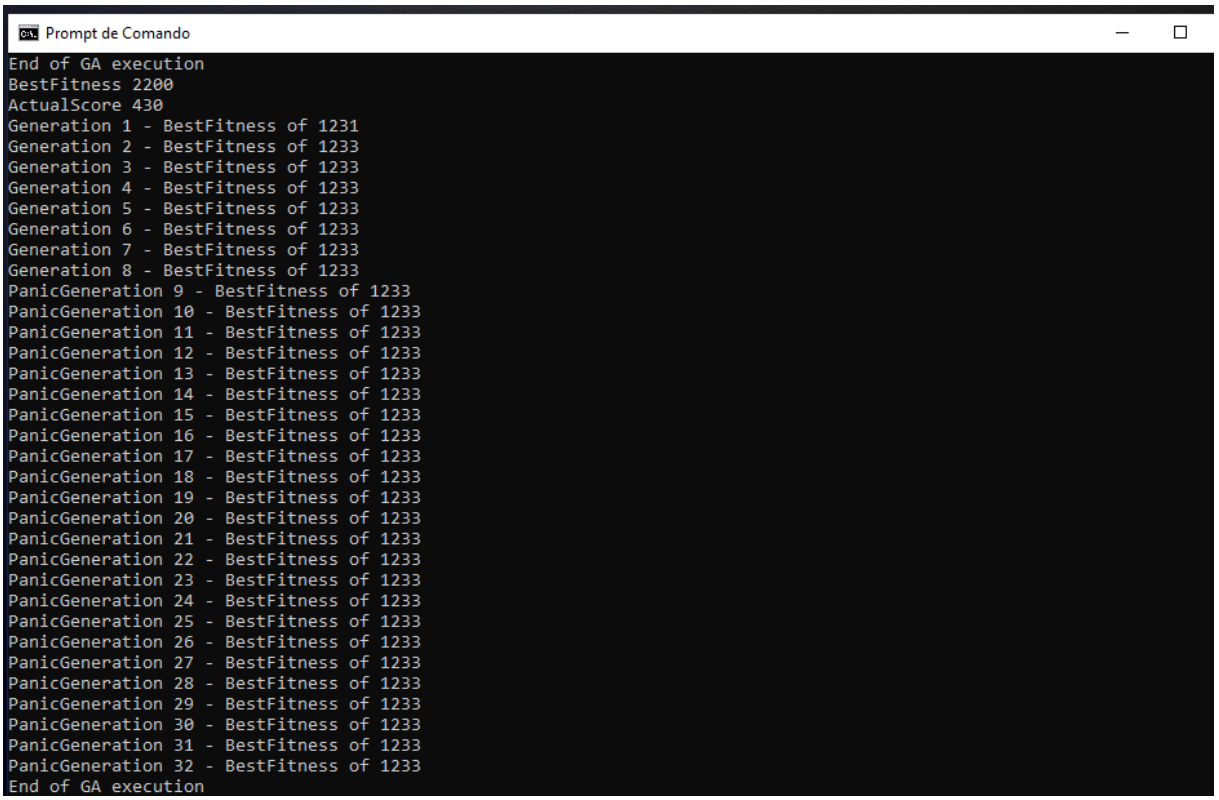
Figura D - Ilustração do Snake com o GA em um instante de tempo mais elevado



Fonte: autoria própria

Deixando o código rodando para a mesma seed, 666, o algoritmo conseguiu manter a cobra viva de modo a obter um score de 430, ou seja, pegaram-se 43 frutinhas. Como mostra o registro presente na Figura E. O qual fora considerado um resultado satisfatório, dentre os demais testes realizados. Tendo o pior caso necessitado de 121 gerações.

Figura E - Melhor resultado obtido deixando o código rodando

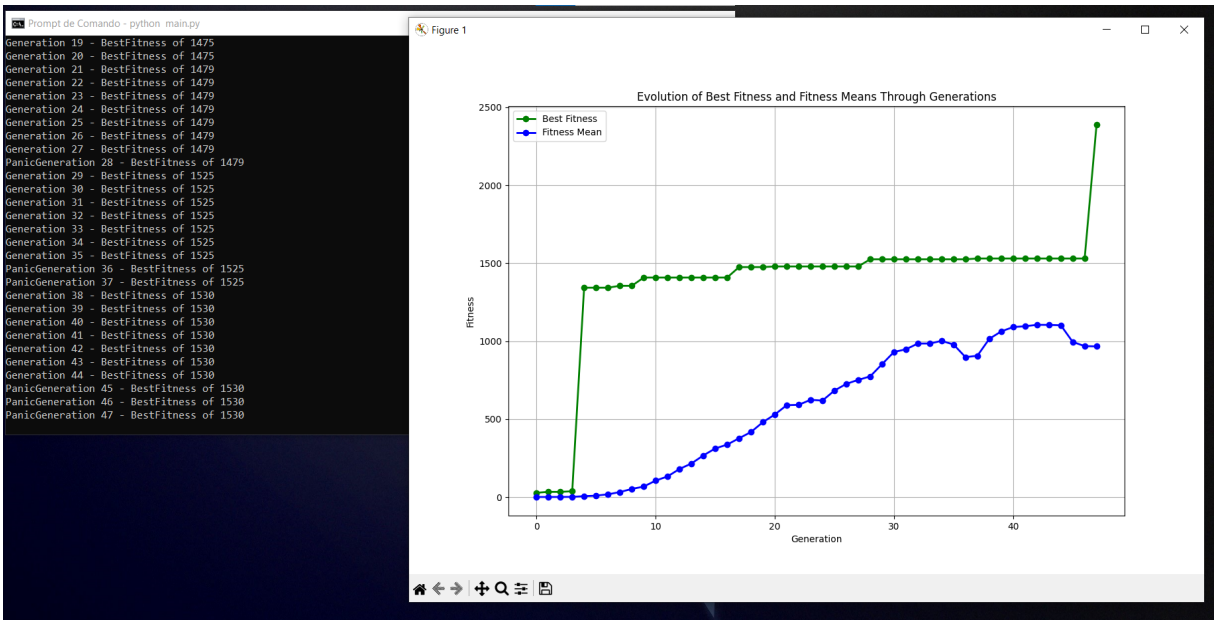


Fonte: autoria própria

A partir dos resultados obtidos, como constatado anteriormente, determinou-se que a abordagem final adotada apresentou resultados satisfatórios.

Nota-se também pela ilustração na Figura F que o algoritmo converge, visto que o valor do maior fitness, bem como a média dos fitness de cada geração cresce ao longo das gerações. Além disso, nota-se a importância do mecanismo de pânico desenvolvido, de modo a livrar o algoritmo dos máximos locais e conseguir encontrar uma solução apropriada.

Figura F - Convergência do algoritmo



Fonte: autoria própria

Ressalta-se que os demais métodos de crossover desenvolvidos apresentaram um desempenho inferior ao crossover de 2 pontos, sendo esse o motivo do mesmo ter sido mantido como principal método.

Além disso, a aplicação do método de seleção de roleta não apresentou um resultado satisfatório, prendendo o GA em um único valor fitness. Quando tal bloqueio não ocorria o resultado observado era mais aleatório que convergente, o que levou a seleção do torneio como método principal.

Outro fator de grande relevância foi a instauração do elitismo. A elite de indivíduos de maior fitness mantida serviu para melhorar a velocidade de convergência do algoritmo e evitar populações que degradavam o fitness geral.

Além disso, a elite de indivíduos serviu como ponto de parada para a execução do algoritmo, permitindo encontrar uma solução mais rapidamente, ou caso considerasse um grande número de membros na mesma, encontrar uma solução mais eficiente para o problema.

5. CONSIDERAÇÕES FINAIS

O desenvolvimento do Algoritmo Genético (AG) para controlar o jogo Snake proporcionou uma experiência enriquecedora, explorando a interseção entre inteligência artificial e entretenimento digital. Ao longo do projeto, foram tomadas decisões cruciais em relação à modelagem do problema, escolha de operadores genéticos e integração com a lógica do jogo.

A modelagem eficiente dos indivíduos no AG, utilizando sequências de genes para representar os movimentos da cobra, demonstrou ser crucial para o desempenho da IA no contexto do Snake. A definição da função de fitness, considerando métricas como tamanho da cobra, tempo de sobrevivência e coleta de alimentos, influenciou diretamente a capacidade do AG de maximizar o desempenho.

A integração bem-sucedida entre o AG e a lógica do jogo Snake destacou a flexibilidade do modelo proposto, permitindo que a IA influenciasse diretamente o comportamento da cobra. A experimentação com diferentes configurações do AG, juntamente com a análise dos resultados em múltiplas execuções, proporcionou insights valiosos sobre a variabilidade e a estabilidade do algoritmo.

6. REFERÊNCIAS

[1] NORVIG, Peter. Inteligência Artificial: Grupo GEN, 2013. E-book. ISBN 9788595156104. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595156104/. Acesso em: 20 de nov. de 2023.

[2] Mantovani, R. **Atividade Prática 03 - Algoritmos Genéticos.** 2023. Disponível em: https://moodle.utfpr.edu.br/pluginfile.php/2925824/mod_resource/content/0/AT03_skane_GA.pdf. Acesso em 28 de nov. de 2023.

[1] R. G. Mantovani, "Intelligent Systems 1 - Activities: Genetic Algorithm," 2023. [Online]. Disponível: https://github.com/rgmantovani/intelligentSystems1/tree/main/activities/03_genetic_algorithm. Acesso em: 20 de nov. de 2023.