Projeto Final - Análise de Algoritmos: Algoritmos Genéticos

Angelo Ferro e Kaio Guilherme

Departamento de Ciência da Computação – Universidade Federal de Roraima

Abstract

This work presents the application of a Genetic Algorithm (GA) to train a simple artificial neural network aimed at controlling an autonomous agent in the classic Snake game environment. The game's deterministic and constrained nature provides an ideal test for evaluating evolutionary algorithms in decision-making and control tasks. The project was developed in two stages: prototyping in Python using the DEAP library, followed by a performance-focused rewrite in C++. Experimental results demonstrate the viability of this approach and the performance gains achieved through implementation optimizations.

Keywords: Genetic Algorithms, Evolutionary Computation, Neural Networks, Autonomous Agents, Optimization, Snake Game, DEAP, C++

I. INTRODUÇÃO

Algoritmos Genéticos (AGs) são uma classe de algoritmos de busca e otimização inspirados nos princípios da evolução natural. Eles têm sido amplamente utilizados em diversas áreas, como engenharia, inteligência artificial e robótica, devido à sua capacidade de encontrar soluções satisfatórias em espaços de busca complexos.

Neste projeto, propomos a aplicação de um AG para o treinamento de uma rede neural artificial simples, com o objetivo de controlar um agente autônomo no ambiente do jogo Snake. O jogo foi escolhido por sua simplicidade e natureza determinística, que o tornam um cenário ideal para avaliar a eficácia de algoritmos evolutivos em tarefas de controle e tomada de decisão.

O desenvolvimento foi realizado em duas etapas: inicialmente, foi feita a prototipagem em Python utilizando a biblioteca DEAP, reconhecida por sua flexibilidade na criação de algoritmos evolutivos. Em seguida, o algoritmo foi reescrito em C++ com o intuito de melhorar o desempenho computacional e permitir análises mais refinadas de eficiência.

II. FUNDAMENTOS TEÓRICOS

A utilização de Algoritmos Genéticos (AGs) como método de otimização baseia-se em princípios inspirados na evolução natural, como seleção natural, recombinação e mutação. Essa técnica envolve a criação e evolução de populações de soluções candidatas ao longo de gerações sucessivas, permitindo a descoberta de soluções robustas para problemas complexos.

Os principais componentes de um AG incluem:

- População: Conjunto de indivíduos representando possíveis soluções para o problema.
- Cromossomo: Codificação de uma solução em forma de cadeia de genes.
- Função de Fitness: Mede o desempenho de cada indivíduo frente ao problema.

Operadores Genéticos:

- Seleção: Escolhe os indivíduos com melhor desempenho para reprodução.
- Crossover (Recombinação): Combina genes de dois pais para formar novos indivíduos.
- Mutação: Introduz pequenas variações aleatórias para manter a diversidade genética.

Este projeto se baseou nos princípios apresentados por Janikow e Clair (1995), adaptando-os para um ambiente simulado semelhante ao jogo Dino Runner, do navegador Chrome. A proposta envolve a aplicação de um AG para treinar uma rede neural artificial (RNA), permitindo que um agente aprenda a se comportar autonomamente no ambiente do jogo.

As redes neurais artificiais são modelos computacionais inspirados na estrutura dos neurônios biológicos. Neste trabalho, utiliza-se uma rede do tipo feedforward com camadas densamente conectadas e função de ativação tangencial hiperbólica (tanh). A integração entre AGs e RNAs permite um processo de aprendizado baseado exclusivamente na avaliação do desempenho dos agentes, sem depender de dados rotulados ou sinais explícitos de reforço.

III. METODOLOGIA

Ambiente de Simulação

O ambiente simula um tabuleiro de dimensões 10x10, onde o agente é uma cobra que inicia com tamanho três em posição central. Uma única maçã é gerada aleatoriamente em uma célula livre a cada coleta. A cada passo, o agente pode realizar uma de três ações: manter a direção atual, virar à esquerda ou virar à direita.

O jogo termina em três situações:

- Colisão com as bordas,
- Colisão com o próprio corpo,
- Expiração do contador de passos.

O contador de passos inicia em 100, é incrementado em 100 a cada coleta de maçã e decrementado em 1 a cada movimento realizado.

Arquitetura da Rede Neural

A rede neural artificial é do tipo feedforward, composta por:

- 4 neurônios de entrada (sensores de proximidade),
- 1 camada oculta com 6 neurônios, ativação tanh,
- 3 neurônios de saída sem ativação (representando as 3 ações possíveis: frente, esquerda, direita).

As entradas são codificadas com base em sensores que verificam:

- Proximidade de paredes,
- Presença de cauda,
- Presença da maçã.

Os pesos da rede são diretamente mapeados para um vetor de DNA, com um total de 42 parâmetros, organizados da seguinte forma:

- 24 pesos entre entrada e camada oculta (6x4),
- 6 bias da camada oculta,
- 12 pesos da camada oculta para a saída (3x4 ou 3x6, com simplificações).

Algoritmo Genético

O AG é utilizado para otimizar os pesos da rede neural, representando cada agente por um vetor de DNA contendo 42 valores contínuos no intervalo [-1, 1]. O processo evolutivo foi configurado com os seguintes parâmetros:

- Tamanho do DNA: 42
- Tamanho da população: 20.000
- Número de gerações: 200
- Proporção de elite: 10%
- Taxa de mutação: 0,005 por gene

O ciclo evolutivo segue as etapas:

- Inicialização da população com DNA aleatório.
- Avaliação de fitness de cada agente em simulação.
- Seleção dos melhores 10%.
- Reprodução por crossover, com aplicação de mutação.
- Repetição do processo por N gerações.

Função de Fitness

A função de fitness utilizada para avaliar o desempenho de cada agente é definida por:

$$\text{fitness} = (\text{maçãs coletadas} \times 100) - \text{passos dados}$$

Esse cálculo busca maximizar a coleta de maçãs enquanto minimiza movimentos desnecessários, incentivando estratégias mais eficientes de navegação no tabuleiro.

IV. RESULTADOS E DISCUSSÃO

Durante os testes experimentais foram utilizados diferentes parâmetros de avaliação

- Configuração 1: 1000 indivíduos por 100 gerações com até 10 maçãs
- Configuração 2: 10000 indivíduos por 100 gerações com até 20 maçãs
- Configuração 3: 20000 indivíduos por 200 gerações com até 30 maçãs

Na primeira versão não havia limitação no número de passos, o que ocasionou rapidamente o surgimento de estratégias subótimas onde os agentes evoluíam para se mover em círculos infinitos com o objetivo de evitar a morte sem buscar a coleta de maçãs

Para mitigar esse comportamento foi introduzido um limite fixo de 50 passos com reinício do contador a cada coleta, o que levou à emergência de uma nova estratégia onde os agentes se mantinham próximos

das paredes a uma distância de um bloco monitorando com os sensores até que uma maçã fosse detectada direcionando-se então para sua coleta. Essa abordagem permitiu crescimento limitado devido ao curto número de ações

Na versão final a limitação foi ajustada para iniciar com 100 passos, sendo incrementados em mais 100 a cada maçã coletada. Essa configuração induziu comportamentos voltados à coleta ativa de maçãs, uma vez que o agente é penalizado por passos e recompensado por ações efetivas. A função de fitness final adotada considerava 100 pontos por maçã válida coletada com penalização proporcional ao número total de passos utilizados, o que promoveu evolução de estratégias eficientes tanto em desempenho quanto em otimização de movimento

Visualização e Diagnóstico

A simulação final é visualizada por meio de renderização com a biblioteca Raylib. São apresentados:

- O tabuleiro com a cobra e a maçã
- Os sensores ativos e direções relativas
- A rede neural do agente com conexões e ativações visíveis
- Painel lateral com estatísticas da execução e desempenho

Essa visualização permite o acompanhamento detalhado do comportamento do agente e facilita o diagnóstico da efetividade da evolução

V. CONCLUSÃO E TRABALHOS FUTUROS

O experimento demonstra a viabilidade de utilizar algoritmos genéticos para o treinamento de redes neurais simples em tarefas de controle. A abordagem se mostrou eficiente para otimizar o comportamento do agente no ambiente restrito do jogo Snake mesmo sem aplicação de técnicas supervisionadas ou aprendizado por reforço clássico. A combinação de simulação determinística, avaliação evolutiva e representação por DNA se apresentou como uma metodologia eficaz para experimentos em inteligência artificial evolutiva

Testes empíricos demonstraram que a performance do agente está diretamente associada aos parâmetros de limitação de passos, número de indivíduos por geração e número de gerações totais. Agentes treinados com maiores populações e ciclos evolutivos mais longos demonstraram comportamentos mais sofisticados e estratégias com maior eficiência exploratória. Como trabalhos futuros propõe-se a introdução de ambientes com obstáculos variáveis, a adição de mecanismos de crossover multi-ponto e a comparação com abordagens baseadas em aprendizado por reforço profundo

PSEUDOCÓDIGO ALGORITMO GENÉTICO

```
Entradas:
   - tamanho_cromossomo: Inteiro
   - função_fitness: Função de avaliação de fitness (recebe um cromossomo e retorna um valor real)
   - minimizar_fitness: Booleano (True para minimizar, False para maximizar)
   - tamanho_população: Inteiro
   - gerações: Inteiro
   - melhor: Real (percentual da população a ser selecionada como elite)
   - probabilidade_mutação: Real (probabilidade de ocorrer uma mutação)

Saídas:
   - população_final: Vetor de cromossomos após o processo evolutivo

Variáveis:
   - população: Vetor de cromossomos
   - fitness_atual: Vetor de valores de fitness para a população
   - geração_atual: Inteiro

Função GerarPopulação() -> Vetor de Cromossomos:
      Iniciar população como vetor vazio
      Para cada i de 1 até tamanho_população faça:
         - Iniciar cromossomo como vetor vazio
         - Para cada j de 1 até tamanho_cromossomo faça:
            - Adicionar valor aleatório no cromossomo
         - Adicionar cromossomo na população
      Retorne população
```

```
Função Seleção(população, fitness, melhor) -> Vetor de Cromossomos:
      Iniciar índice como vetor de índices da população
      Se minimizar_fitness então:
         - Ordenar índice com base no fitness em ordem crescente
      Senão:
         - Ordenar índice com base no fitness em ordem decrescente
      Definir n como o maior valor entre 1 e tamanho_população * melhor
      Iniciar população_selecionada como vetor vazio
      Para i de 1 até n faça:
         - Adicionar população[indice[i]] em população_selecionada
      Retorne população_selecionada

Função Cruzamento(p1, p2) -> Cromossomo:
   - Escolher ponto de corte aleatório entre 1 e tamanho_cromossomo - 1
   - Iniciar filho como vetor vazio
   - Para i de 1 até ponto de corte faça:
         - Adicionar p1[i] em filho
      Para i de ponto de corte + 1 até tamanho_cromossomo faça:
         - Adicionar p2[i] em filho
      Retorne filho

      Função Mutação(cromossomo):
            Para cada gene em cromossomo faça:
               - Gerar valor aleatório probabilidade_mutação
               - Se o valor aleatório for menor que probabilidade_mutação então:
                     - Substituir gene por valor aleatório
            Retorne cromossomo
Função AvançarGeração() -> Booleano:
      Se população estiver vazia então:
            Retorne FALSO

   - Calcular fitness de todos os indivíduos da população
   - Se geração_atual >= gerações então:
            Retorne FALSO

   - Selecionar elite utilizando a função Seleção
   - Iniciar nova_população com a elite
   - Enquanto tamanho da nova_população < tamanho_população faça:
         - Selecionar dois pais aleatórios da elite
         - Realizar cruzamento entre os pais para gerar filho
         - Aplicar mutação no filho
         - Adicionar filho em nova_população

   - Garantir que nova_população tenha tamanho igual a tamanho_população
   - Atualizar população com nova_população
   - Incrementar geração_atual
            Retorne VERDADEIRO

Início do Algoritmo:
   - Gerar população inicial chamando GerarPopulação
   - Enquanto o número de gerações não for atingido faça:
         - Chamar AvançarGeração
         Retorne população final
```

REFERÊNCIAS

[1] C. Janikow and D. Clair, “Simulating Nature’s Methods of Evolving the Best Design Solution,” IEEE, 1995.

[2] DEAP: Distributed Evolutionary Algorithms in Python. [Online]. Available: https://github.com/DEAP/deap

[3] I. Seidel, “Genetic Algorithm - The Nature of Code (Part 1),” YouTube, Apr. 28, 2020. [Online]. Available:
Artificial Intelligence in Google's Dinosaur (E…

[4] Angelox99, "FinalProject_DCC606_Tema_Genetic_Algorithms_RR_2025," GitHub, 2025. [Online]. Available: https://github.com/Angelox99/FinalProject_DCC606_Tema_Genetic_Algorithms_RR_2025