Aplicação de Algoritmos Genéticos na Solução do Problema das N-Rainhas

Jessé Santana Veloso[1]

[1]Curso de Bacharelado em Ciência da Computação -
Universidade Estadual do Centro-Oeste (Unicentro)
Guarapuava-PR-Brasil

jessesantanaveloso23@gmail.com

Resumo. *O desafio das N-rainhas requer a disposição de n rainhas em um tabuleiro de xadrez com tamanho N, de maneira que nenhuma delas ataque a outra. Este problema, apesar de possuir mais de 90 soluções diferentes, apresenta uma complexidade considerável devido à ampla mobilidade da rainha. Este artigo se concentra na aplicação de Algoritmos Genéticos (AG), utilizando mecanismos inspirados na evolução natural que Charles Dawin propôs, como seleção, cruzamento e mutação para a busca pela otimização. Além do uso de aprendizado de máquina baseado em reforço para a seleção de operadores genéticos de cruzamento para o AG. Ao fim desse trabalho o algoritmo se mostrou eficiênte na busca de soluções com sucesso em poucas gerações.*

Palavras Chave: Xadrez, Otimização, Inteligência Artificial.

1. Introdução

O problema das *N*-rainhas é um intrigante desafio matemático que investiga as diversas disposições possíveis para posicionar *n* rainhas em um tabuleiro de xadrez com dimensões *n* x *n*. O objetivo principal é garantir que nenhuma rainha possa atacar outra, o que implica que não deve haver coincidências de posições entre elas nas mesmas colunas, linhas ou diagonais. Esse problema pode ser melhor compreendido através da representação visual de uma configuração de rainhas em um tabuleiro, conforme ilustrado na Figura 1.

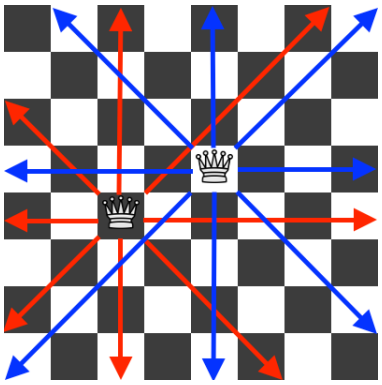

Figura 1: Movimentação da rainha no xadrez [Solarian 2017]

A origem desse enigma remonta a um desafio proposto de forma anônima, focando no tabuleiro de xadrez padrão, que possui dimensões 8 x 8. Esse desafio matemático foi

associado ao jogador de xadrez Max Bezzel em 1848 ,mas ganhou atenção considerável após Franz Nauck, em 1850, ter escrito uma carta ao astrônomo Schumacher, mencionando suas observações sobre as soluções desse enigma, posteriormente extendeu o problema para o tamanho *n*. [Campbell 1977].

Nesse contexto, o renomado matemático Carl Friedrich Gauss se deparou com o problema e trabalhou em suas soluções, conjecturando que existiriam 72 configurações possíveis para as 8 rainhas. Contudo, essa afirmação foi rapidamente contestada, uma vez que foi comprovado que existiam, na verdade, 92 soluções distintas para o problema. Este número inclui um conjunto de 12 soluções fundamentais, que são únicas e não correspondem a rotações ou reflexos de outras configurações. Essas 12 soluções são ilustradas na Figura 2 e representam as diferentes maneiras pelas quais as rainhas podem ser dispostas no tabuleiro sem se atacarem mutuamente [Rivin and Vardi 1994].

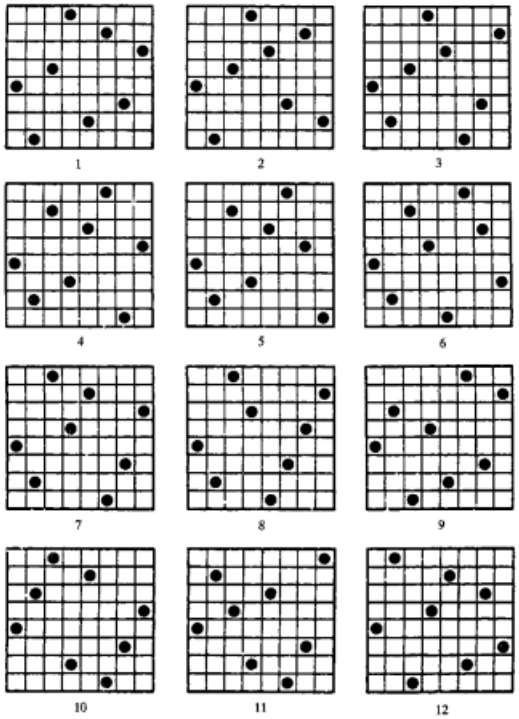

Figura 2: Soluções fundamentais em um tabuleiro 8 x 8 [Rivin and Vardi 1994]

Com o avanço da ciência e da tecnologia, a criação de computadores capazes de resolver problemas antes solucionados apenas por seres humanos tornou-se uma realidade. Um exemplo notável disso é o desenvolvimento de Inteligências Artificiais (IA), uma área da ciência da computação que busca entender o funcionamento do cérebro humano e capacitar os computadores a raciocinar de forma semelhante.

Diversas técnicas de IA, como afirmado por Rich (1993, p. 9), são métodos que buscam o conhecimento em contextos específicos, propondo soluções viáveis. Essas técnicas podem ser aplicadas em inúmeras situações e devem fornecer uma estrutura que facilite a correção de erros. Entre as técnicas mais populares estão os Algoritmos Genéticos (AG), utilizados neste trabalho. Para otimizar o funcionamento da AG, foi utilizada a seleção de operadores de cruzamento durante a execução do algoritmo, a abordagem de *Multi-Armed Bandits* (MABs), juntamente com o método *Upper*

Confidence Bound (UCB), foi empregada. Essa estratégia adaptativa permite selecionar dinâmicamente o operador de cruzamento mais eficaz, aumentando a eficiência do processo evolutivo e melhorando os resultados na resolução do problema das *N*-rainhas [Peruci 2024].

2. Algoritmos Genéticos

Um Algoritmo Genético (AG) é uma técnica de Inteligência Artificial (IA) que busca otimizar a solução de problemas complexos. Esta técnica tem seus conceitos e fundamentos baseado na genética e nas teorias de evolução e de seleção natural propostos por Charles Darwin [Rivin and Vardi 1994]. Os AGs são compostos majoritariamente por:

- Indivíduo: Uma possível solução parcial para o problema, também chamada de cromossomo, podendo ser representadas por *strings*, vetores, listas, dentre outras estruturas de dados.
- População: É um conjunto total de indivíduos.
- Operadores Genéticos: São funções que se aplicam à população para se obter uma nova população.
- Função de aptidão: Também conhecida como função de *fitness* se trata do cálculo que irá guiar o algoritmo para se obter novas populações e parâmetro dos operadores.

Para compreensão da AG pode-se fazer o seguinte exemplo de execução:

Primeiramente gera-se, aleatoriamente, um conjunto de indivíduos que podem ser possíveis soluções para o problema. Então, são selecionados os indivíduos pais que irão gerar novos cromossomos através de algum critério, e então ocorre o *crossover*. O *crossover* será o ato de, com base nos valores dos cromossomos pais, gerar um novo indivíduo que contenha informações de ambas partes. Após isso poderão ocorrer as mutações, que, ao contrário do *crossover* são modificações menores, geralmente alterando apenas um dos valores dentro do cromossomo [Luiz 2012].

2.1. Indivíduo (Cromossomo)

Em um AG para resolver o problema das *N*-rainhas, cada indivíduo, ou cromossomo, representa uma possível solução para o problema. Cada posição da coluna do tabuleiro é representada por um número em um vetor, onde o índice do vetor corresponde à posição x e ao valor do índice à posição y de uma rainha no tabuleiro, representado na Figura 3. Cada valor no vetor foi feito único, garantindo que nenhuma rainhas esteja na mesma linha [Luiz 2012].

Por exemplo, para um tabuleiro de 8 x 8 em que os valores variam entre 0 e 7, uma solução possível poderia ser representada pelo vetor [2, 0, 3, 1, 4, 6, 7, 5]. Isso significa que a primeira rainha está na terceira posição da primeira coluna, a segunda rainha está na primeira posição da segunda coluna, e assim por diante.

O tamanho do cromossomo sempre depende de n. À medida que se aumenta o tamanho do tabuleiro, o número de disposições possíveis também cresce de maneira exponencial. O cálculo para o valor de todas as disposições possíveis no tabuleiro é dado por $N!$.

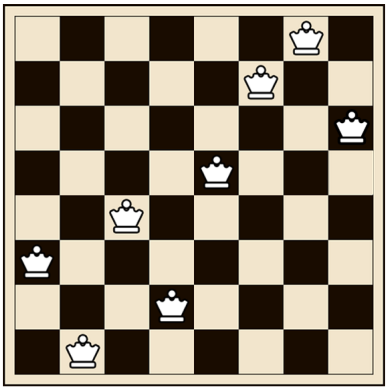

Figura 3: Exemplo de vetor em um tabuleiro 8 x 8

Para determinar o número total de soluções possíveis, não existe uma fórmula geral para todos os valores de n, mas o problema pode ser abordado matemáticamente por meio de métodos combinatórios e teoria de grafos. A maneira mais comum de efetuar esse cálculo é utilizando um algoritmo de *backtracking*, onde você coloca a rainhas em uma linha e tenta colocar as outras nas linhas subsequentes garatindo que não haja ataques, embora isso se torne complexo [Sarkar and Nag 2023].

Soluções conhecidas:

- N = 1: 1 solução
- N = 2: 0 soluções
- N = 3: 0 soluções
- N = 4: 2 soluções
- N = 5: 10 soluções
- N = 6: 4 soluções
- N = 7: 40 soluções
- N = 8: 92 soluções

2.2. Função de Aptidão (*Fitness*)

Assim como em um ambiente real, a aptidão é desenvolvida de acordo com o ambiente em que o indivíduo está presente, logo, se ele não estiver apto a viver em determinado ambiente, rapidamente será extinto. Portanto deve-se atentar para os indivíduos mais propícios a constituir a solução do problema [Rosa 2009].

Segundo Rosa (2009), esta é a única etapa de Algoritmo Genético que não é genética. Essa etapa é o elo que une a AG e o problema específicado, sendo que a função de aptidão deve ser específica para cada problema ”[...]a função de aptidão é definida de acordo com outros membros da atual população em um algoritmo genético [Rosa and Luz 2009].

Para a configuração de N-rainhas, a função de aptidão é definida pela contagem de conflitos presentes no tabuleiro. A função calcula o número de pares que estão em posições de ataque, que inclui ataques nas mesmas linhas, colunas ou diagonais e então subtrai do valor máximo de colisões que podem ocorrer no tabuleiro de tamanho n. O

cálculo do valor máximo de colisões possíveis em um tabuleiro de tamanho *n* é exibido pela seguinte equação:

$$\text{Valor máximo de colisões possíveis} = \frac{N(N-1)}{2} \tag{1}$$

Cada coluna do tabuleiro conterá apenas uma rainha, tornando irrelevante a comparação entre rainhas na mesma coluna. Além disso, o algoritmo desenvolvido garante que os números no vetor sejam únicos, eliminando a necessidade de verificar colisões entre rainhas na mesma linha. Dessa forma, o foco principal da implementação se torna o cálculo das colisões nas diagonais, representado por:

$$\text{Total de colisões diagonais} = \sum_{\text{pares de rainhas}} \text{se } (i+j \text{ ou } i-j \text{ são iguais, então } 1, \text{ caso contrário } 0) \tag{2}$$

A Figura 4 mostra um pseudocódigo simplificado da função *fitness*, retornando a subtração do número de colisões atuais do valor máximo de colisões. Tornando assim indivíduos com um maior fitness, indivíduos melhores com menos conflitos.

```
Função fitness(cromossomo)
    colisoes += colisoes na diagonal

    return Total de colisões possiveis - colisoes
```

Figura 4: Pseudocódigo do cálculo do fitness

A Figura 5 ilustra um exemplo do cálculo do *fitness* em um tabuleiro 8 x 8. A linha amarela destaca o retângulo em análise, onde se verifica se a rainha nessa linha colide com alguma outra. Para evitar contagens duplicadas, a rainha em questão é comparada apenas com os elementos que estão à sua direita.

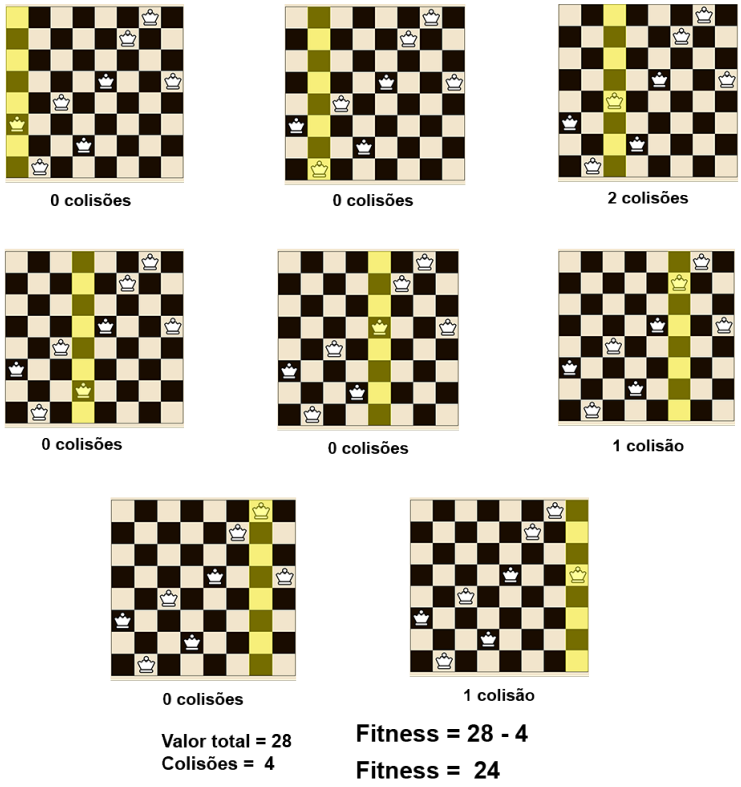

Figura 5: Exemplo do funcionamento e cálculo do *fitness*

2.3. Cruzamento (*Cross-over*)

Nesta etapa, indivíduos são selecionados aleatoriamente, conforme a taxa de *crossover* especificada. Em seguida, os cromossomos de cada par de indivíduos selecionados são particionados em um ou mais pontos, conhecidos como pontos de corte. Assim como na biologia, é durante essa fase de cruzamento que ocorre a troca de informações genéticas [Rosa 2009]. Para este trabalho, três métodos comuns de realizar essa troca foram escolhidos:

1. *Uniform Crossover* (ULX): Neste método, cada gene do pai é selecionado aleatoriamente para ser herdado de um dos pais, resultando em uma combinação aleatória dos genes dos dois progenitores determinada por uma máscara que escolhe qual característica o filho herdará [Peruci 2024].

Figura 6: Uniform *Crossover* [Chaudhry and Usman 2017]

2. *One Point Crossover* (OPX): Nesse caso, um único ponto de corte é escolhido ao longo dos cromossomos. Os genes de um dos pais são trocados a partir desse

ponto, gerando novos filhos que combinam características de ambos os progenitores [Peruci 2024].

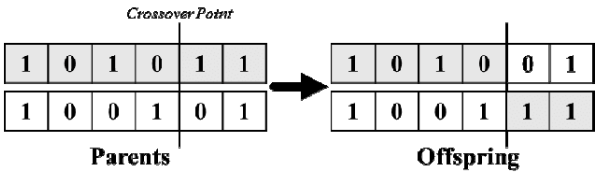

Figura 7: One Point *Crossover*[Abad et al. 2015]

3. *Order Based Crossover* (OBX): Este método é particularmente útil em problemas de otimização onde a ordem dos elementos é importante, como em problemas de roteamento. Ele seleciona uma subsequência de um dos pais e a insere em um dos filhos, preenchendo os demais espaços com os elementos do outro pai, mantendo a ordem relativa dos genes [Peruci 2024].

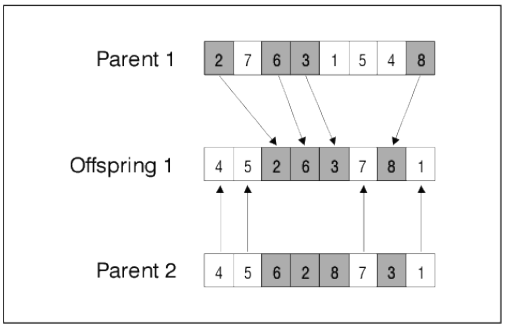

Figura 8: Order Based *Crossover* [Shariff et al. 2013]

2.4. *Multi-armed bandtis* e *Upper Confidence Bound*

O problema conhecido como *Multi-armed Bandit* (MAB) pode ser visto como um cenário em que um agente, representado como um jogador, busca maximizar suas recompensas. Essa situação é frequentemente comparada a uma máquina de caça-níqueis com várias alavancas em um cassino. [Peruci 2024] Cada alavanca corresponde a um ”braço”da máquina, e o objetivo do jogador é acumular a maior quantidade de dinheiro possível, puxando as alavancas ao longo do tempo. Em cada etapa, o jogador seleciona um braço e recebe uma recompensa. Com base nos resultados obtidos, ele consegue aprender quais máquinas oferecem as melhores chances de retorno financeiro, exemplificado na Figura 9 [Kuleshov and Precup 2014].

Upper Confidence Bound (UCB) refere-se a uma categoria de algoritmos eficazes que abordam o dilema entre exploração e exploração em problemas associados a bandits e aprendizado por reforço (Hao et al., 2019). Existem várias variantes disponíveis, como UCB-1, UCB-2 e UCB-TUNED, cada uma com características específicas [Auer et al. 2002].

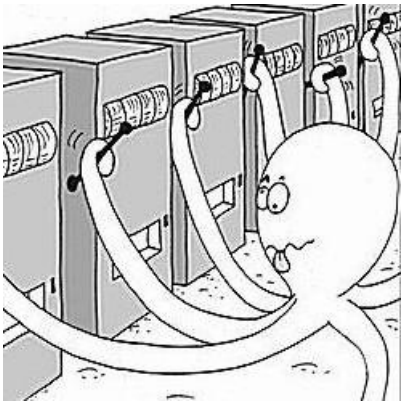

Figura 9: Exemplo clássico do ”Polvo no cassino”que representa simplificadamente o funcionamento do *Multi-armed Bandit* [PAP]

Na implementação do algoritmo, inicia-se configurando um número fixo de opções, representadas pelos operadores de crossover. Para cada um deles, o algoritmo mantém um registro da frequência com que é selecionado, bem como da soma das recompensas associadas a essas escolhas.

Inicialmente, o algoritmo assegura que cada opção seja escolhida pelo menos uma vez, permitindo a coleta de informações sobre suas respectivas recompensas. Após essa fase exploratória, o algoritmo calcula uma estimativa da recompensa média para cada opção e ajusta essa estimativa com um fator de confiança, que diminui à medida que mais escolhas são realizadas. Isso possibilita que o algoritmo não se baseie apenas em opções que já demonstraram sucesso, mas também considere novas alternativas [Peruci 2024].

Após selecionar um ”braço”, o algoritmo registra a recompensa obtida e utiliza essa informação para atualizar a soma total de recompensas. Esse processo se repete, permitindo uma iteração constante entre exploração e aproveitamento das opções, enquanto as estimativas de recompensa são gradualmente refinadas com base no aprendizado por reforço [Peruci 2024].

2.5. Mutação

A mutação é um conceito fundamental nos algoritmos genéticos. Em termos simples, a mutação refere-se a alterações aleatórias que ocorrem nas soluções de uma população, com o objetivo de introduzir diversidade genética. Essa variação é essencial para evitar que o algoritmo se prenda em ótimos locais, permitindo que explore novas possibilidades e melhore continuamente as soluções. Através da mutação, o algoritmo pode escapar de soluções subótimas e se adaptar melhor às complexidades do problema em questão [Sarkar and Nag 2023].

Na função implementada, foi utilizada uma função que seleciona aleatóriamente dois alelos (ou elementos) da sequência e os troca de posição. Este processo simples é a base para o que é chamado de mutação simples.

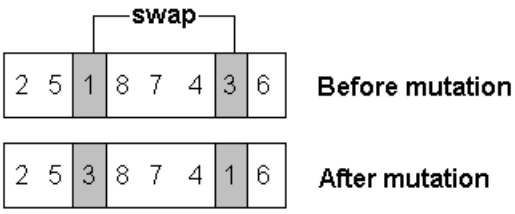

Figura 10: Exemplo de mutação [El Majdoubi et al. 2020]

2.6. Funcionamento

A implementação do algoritmo foi realizada na linguagem Python[1], utilizando algumas de suas bibliotecas populares, como NumPy[2] e Matplotlib[3]. O ambiente virtual escolhido para a execução do código foi a plataforma Google Colab[4], que oferece recursos de processamento na nuvem.

A implementação começa selecionando o número de pais com base na taxa de cruzamento. Esses pais são então adicionados à nova população de indivíduos. Em seguida, os pais previamente selecionados são escolhidos aleatoriamente para realizar o *crossover*, com o operador selecionado pelo *Multi-Armed Bandi*t (MAB), gerando novos filhos e executando a mutação até que 50% da nova população esteja completa. Após essa etapa, o vetor da população é preenchido com indivíduos aleatórios até que a nova população atinja seu tamanho total. A Figura 11 mostra seu funcionamento em pseudocódigo..

```
INICIAR nova_populacao COMO lista vazia
INICIAR pais COMO lista vazia

# Determina quantos pais vão participar do crossover
num_pais = taxa_cruzamento * tamanho_populacao

# Seleciona os pais para a nova população
ENQUANTO tamanho(pais) < num_pais FAÇA
    ADICIONAR selecaoAleatoria(populacao) A pais

# Preenche a nova população com os pais selecionados
ADICIONAR pais A nova_populacao

# Preenche o restante da nova população com os filhos
ENQUANTO tamanho(nova_populacao) < tamanho_populacao FAÇA
    pai1 = selecaoAleatoria(pais)
    pai2 = selecaoAleatoria(pais)
    filho = crossover(pai1, pai2)
    mutacao(filho)
    ADICIONAR filho A nova_populacao

# Ajusta a nova população se for maior que o esperado
SE tamanho(nova_populacao) > tamanho_populacao ENTÃO
    REMOVER um indivíduo aleatório de nova_populacao

# Preenche a nova população com indivíduos aleatórios se necessário
ENQUANTO tamanho(nova_populacao) < tamanho_populacao FAÇA
    ADICIONAR gerarCromossomo() A nova_populacao

# Atualiza a população antiga pela nova
populacao = nova_populacao
```

Figura 11: Pseudocódigo do algoritmo [El Majdoubi et al. 2020]

[1]Python: Linguagem de programação de alto nível, amplamente utilizada para desenvolvimento web, automação, análise de dados e aprendizado de máquina, conhecida por sua sintaxe clara e legibilidade.

[2]NumPy: Biblioteca fundamental para a computação científica em Python, fornecendo suporte para arrays e operações matemáticas eficientes.

[3]Matplotlib: Biblioteca amplamente utilizada para a criação de visualizações gráficas em Python, permitindo a geração de gráficos e plots.

[4]Google Colab: Plataforma de notebooks que permite a execução de código Python no navegador, com acesso a GPUs e armazenamento no Google Drive.

3. Resultados

Segue aqui os resultados dos testes realizados, considerando os valores padrão como: tamanho de tabuleiro = 8, taxa de cruzamento = 0.7, taxa de mutacao = 0.1, tamanho da população = 10 [Rivin and Vardi 1994]. Os testes realizados variaram um dos respectivos valores enquantos os outros continham o valor padrão, sendo feitos 5 testes alterando os valores de cada variável. Vale ressaltar que o número de gerações não é algo fixo, já que o algoritmo irá parar apenas quando encontrar a solução, tornando assim as soluções com menores números de gerações melhores.

Tabela 1: Resultados do Modelo em Função do tamanho do tabuleiro

Resultados					
Tamanho do tabuleiro	Possibilidades de tabuleiro	Taxa de cruzamento	Taxa de mutação	Tamanho da população	Geração da solução
8	40320	0.7	0.1	10	121
9	362880	0.7	0.1	10	203
10	3628800	0.7	0.1	10	4201
11	39916800	0.7	0.1	10	999
12	479001600	0.7	0.1	10	2627

Tabela 2: Resultados do Modelo em Função da taxa de *crossover*

Resultados					
Tamanho do tabuleiro	Possibilidades de tabuleiro	Taxa de cruzamento	Taxa de mutação	Tamanho da população	Geração da solução
8	40320	0.7	0.1	10	127
8	40320	0.6	0.1	10	17
8	40320	0.5	0.1	10	100
8	40320	0.4	0.1	10	594
8	40320	0.3	0.1	10	1581

Tabela 3: Resultados do Modelo em Função da taxa de mutação

Resultados					
Tamanho do tabuleiro	Possibilidades de tabuleiro	Taxa de cruzamento	Taxa de mutação	Tamanho da população	Geração da solução
8	40320	0.7	0.1	10	127
8	40320	0.7	0.2	10	15
8	40320	0.7	0.3	10	926
8	40320	0.7	0.4	10	439
8	40320	0.7	0.5	10	487

Tabela 4: Resultados do Modelo em Função do tamanho da população

Resultados					
Tamanho do tabuleiro	Possibilidades de tabuleiro	Taxa de cruzamento	Taxa de mutação	Tamanho da população	Geração da solução
8	40320	0.7	0.1	10	127
8	40320	0.7	0.1	20	2
8	40320	0.7	0.1	30	56
8	40320	0.7	0.1	40	10
8	40320	0.7	0.1	50	25

A análise dos testes realizados revela que, na maioria das execuções, a variância dos valores obtidos é significativa. Isso se deve à ampla gama de possibilidades apresentadas pelos tabuleiros, onde, em algumas situações, um dos indivíduos na primeira geração já representa a solução ideal.

Os resultados também indicam que a variável que mais impactou a qualidade das soluções foi o tamanho da população. Quanto maior o número de indivíduos gerados em cada geração, maior é a probabilidade de encontrar um cromossomo que corresponda ao resultado esperado. Essa relação destaca a importância de um dimensionamento adequado da população para otimizar o desempenho do algoritmo.

4. Conclusões

O algoritmo para a solução das N-rainhas demonstrou ser uma abordagem eficaz para resolver o clássico problema de posicionamento de rainhas em um tabuleiro, garantindo que nenhuma rainha ataque a outra. Ao longo do estudo, foi implementado uma série de técnicas de otimização como algoritmos genéticos e aprendizado por reforço, que se mostraram promissores na busca por soluções eficientes.

Os resultados obtidos indicam que as estratégias adotadas, como seleção, cruzamento e mutação, apesar de inferirem uma alta variância de resultados, melhoraram significativamente a qualidade do algoritmo. Em vista que a capacidade do algoritmo de explorar diferentes combinações e evitar ótimos locais foi um fator crucial.

Em suma, o uso de algoritmos genéticos para o problema das N-rainhas não apenas confirma a viabilidade desta abordagem, mas também abre portas para a aplicação de técnicas semelhantes em problemas mais complexos de otimização.

Referências

Multi-armed bandits. `https://paperswithcode.com/task/multi-armed-bandits`. Acesso em: 21 out. 2024.

Abad, R. P., Fillone, A., Dadios, E., and Roquel, K. I. D. (2015). An application of genetic algorithm in optimizing jeepney operations along taft avenue, manila. pages 1–6.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256.

Campbell, P. J. (1977). Gauss and the eight queens problem: A study in miniature of the propagation of historical error. *Historia Mathematica*, 4(4):397–404.

Chaudhry, I. A. and Usman, M. (2017). Integrated process planning and scheduling using genetic algorithms. *Technical gazette*, 24:1401–1409.

El Majdoubi, O., Abdoun, F., Najat, R., and Otman, A. (2020). Artificial intelligence approach for multi-objective design optimization of composite structures: Parallel genetic immigration.

Kuleshov, V. and Precup, D. (2014). Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*.

Luiz, A. (2012). Algoritmos genéticos. Apostila EPAC – Encontro Paranaense de Computação.

Peruci, F. T. (2024). Problema quadrático de alocação aplicado ao layout hospitalar usando aprendizado por reforço. In *Anais da II Conferência Internacional de Políticas Públicas e Ciência de Dados*, Curitiba (PR). UTFPR. Acesso em: 21/10/2024.

Rivin, I. and Vardi (1994). The n-queens problem. *The American Mathematical Monthly*, 101(7):629–639.

Rosa and Luz, H. S. (2009). Conceitos básicos de algoritmos genéticos: Teoria e prática. In *Anais do XI Encontro de Estudantes de Informática do Tocantins*, page 11, Palmas. CEULP/ULBRA.

Sarkar, U. and Nag, S. (2023). An adaptive genetic algorithm for solving nqueens problem. *Jadavpur University*.

Shariff, S., Moin, N., and Omar, M. b. (2013). An alternative heuristic for capacitated p-median problem (cpmp). pages 916–921.

Solarian, P. (2017). Eight queens puzzle in python. Acesso em: 23 out. 2024.