



UEPB

Universidade Estadual da Paraíba

Departamento de Computação – DC

Inteligência Artificial – 2025.1

Professor: Wellington Candeia de Araújo

DJHONATAH WESLEY C. ALVES

PROBLEMA DAS N-RAINHAS COM BLOQUEIOS

CAMPINA GRANDE – PB

04 de maio de 2025

DJHONATAH WESLEY CAVALCANTI ALVES

PROBLEMA DAS N-RAINHAS COM BLOQUEIOS

Relatório apresentado no curso de Ciência da Computação da Universidade Estadual da Paraíba e na disciplina de Inteligência Artificial, referente ao período 2025.1

Professor: Wellington Candeia de Araújo

CAMPINA GRANDE – PB

04 de maio de 2025

RESUMO

Este relatório apresenta a aplicação de um Algoritmo Genético Generativo para resolver o Problema das N-Rainhas com posições bloqueadas, uma variação do clássico problema das N-Rainhas. Casas do tabuleiro são bloqueadas aleatoriamente (7% a 13% do total) e o objetivo é posicionar n rainhas sem conflitos entre si e sem ocupar casas bloqueadas. A abordagem evolutiva empregou operadores de seleção por torneio ($k=3$), crossover, mutação (probabilidade 0,1 por gene), mini-repair e elitismo, com critério de parada em solução ótima ou 2000 gerações. Testes foram realizados para $n = 1, 2, 5, 8, 9, 10, 12, 16$ e 20 , demonstrando robustez e escalabilidade do método

Palavras-chave: Algoritmo Genético; N-Rainhas; Bloqueios; MiniRepair; Elitismo.

INTRODUÇÃO

O problema das N-Rainhas consiste em posicionar n rainhas num tabuleiro $n \times n$ de modo que nenhuma ataque outra (linhas, colunas e diagonais). Na variação com bloqueios, algumas casas do tabuleiro são indisponíveis, reduzindo o espaço de busca e aumentando a complexidade da solução. Nesta atividade, substituímos técnicas de busca exaustiva por um Algoritmo Genético Generativo, capaz de explorar um leque amplo de soluções mediante recombinação e mutações, aproximando-se de soluções válidas de forma eficiente mesmo em cenários com restrições.

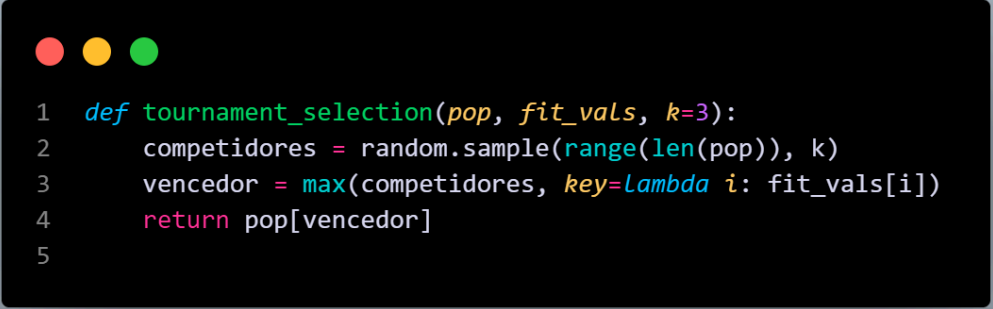
OBJETIVO

Implementar e avaliar um Algoritmo Genético Generativo para o Problema das N-Rainhas com posições bloqueadas, analisando desempenho em diferentes tamanhos de tabuleiro quanto a geração de solução, número de gerações necessárias e tempo de execução.

METODOLOGIA

Como o Algoritmo Genético **não funciona como uma busca em árvore**, ele não usa heurística explícita, mas sim utiliza de operadores evolutivos:

Seleção: por torneio ($k = 3$), onde os melhores indivíduos são escolhidos para que a partir deles haja indivíduos cada vez melhores.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a Python function definition for tournament selection.

```
1 def tournament_selection(pop, fit_vals, k=3):
2     competidores = random.sample(range(len(pop)), k)
3     vencedor = max(competidores, key=lambda i: fit_vals[i])
4     return pop[vencedor]
5
```

Crossover: Ele combina partes de dois indivíduos (pais) para gerar novos indivíduos (filhos), com a ideia de **recombinar boas características** dos pais.

```
1 def crossover(pai1, pai2, pc):
2     n = len(pai1)
3     if random.random() < pc:
4         ponto = random.randint(1, n-1)
5         return pai1[:ponto] + pai2[ponto:], pai2[:ponto] + pai1[ponto:]
6     return pai1[:], pai2[:]
```

Mutação: com probabilidade 0.1 por gene, sorteando uma nova coluna válida (não bloqueada).

```
1 def mutacao(individuo, bloqueios, pm):
2     n = len(individuo)
3     for i in range(n):
4         if random.random() < pm:
5             individuo[i] = random.choice([c for c in range(n) if (i, c) not in bloqueios])
6     return individuo
```

MiniRepair: É uma técnica usada em algoritmos genéticos para corrigir parcialmente indivíduos da população que possuem muitos conflitos (ou seja, soluções ruins).

Elitismo: É uma estratégia nos algoritmos genéticos onde os melhores indivíduos da geração atual são automaticamente copiados para a próxima geração sem alterações.

```
1 pop_sorted = [ind for _, ind in sorted(zip(fit_vals, pop), key=lambda x: x[0], reverse=True)]
2 nova_pop = pop_sorted[:elitismo_k] # preserva elites
```

Ambiente de Execução:

Tabela 01: Características principais do computador utilizado.

Placa Mãe	B550M - AORUS ELITE
Processador	AMD Ryzen 5 5600G @ 3.90GHz
Memória RAM	16GB
Sistema Operacional	Windows 11
Ambiente de Execução	Visual Studio Code – Python 3.10

Fonte: Autoria Própria.

ESTRUTURA DO CÓDIGO

O código foi estruturado em funções principais dentro de um único arquivo Python:

- **gerar_tabuleiro_com_bloqueios_melhorado(n, seed=42):** gera o tabuleiro $n \times n$ com bloqueios aleatórios entre 7% e 13% do total de casas, garantindo no máximo $n-1$ bloqueios por linha.
- **inicializar_populacao(n, bloqueios, tam_pop=200):** cria a população inicial com indivíduos válidos (sem rainhas em casas bloqueadas).
- **fitness(individuo):** avalia o indivíduo pelo número de pares de rainhas não conflitantes.

- **conflitos(individuo)**: calcula o número de conflitos entre rainhas.
- **minirepair(individuo, bloqueios, max_iters=5)**: ajusta até 5 posições problemáticas tentando reduzir conflitos.
- **tournament_selection(pop, fit_vals, k=3)**: operador de seleção por torneio.
- **crossover(pai1, pai2, pc=0.8)**: operador de recombinação com probabilidade 0,8.
- **mutacao(individuo, bloqueios, pm=0.2)**: operador de mutação com probabilidade 0,2.
- **algoritmo_genetico(...)**: função principal que executa o algoritmo genético com os operadores e parâmetros descritos.
- **main()**: executa os testes para os valores de n especificados e imprime os resultados.

RESULTADOS E DISCUSSÕES

Tabela 02: Resultados obtidos com os testes realizados com o algoritmo Generativo.

N	SOLUÇÃO ENCONTRADA	GERAÇÕES USADAS	TEMPO (MS)
1	SIM	1	0.17
2	NÃO	2000	5469.32
5	SIM	1	1.01
8	SIM	2	31.67
9	SIM	2	42.32
10	SIM	2	55.05
12	SIM	3	174.44
16	SIM	57	10533.97
20	SIM	175	61804.49

Fonte: Autoria Própria.

Foi observado que mesmo com 2000 gerações para o tamanho $n=2$ não foi possível encontrar solução, o que é consistente com a teoria: não há solução para 2 rainhas que não se ataquem em um tabuleiro 2×2 . O algoritmo genético se mostrou robusto e completo, especialmente com o uso de elitismo e minireparo, permitindo encontrar soluções rapidamente para tamanhos de tabuleiro grandes $n=20$, por exemplo. A abordagem evolutiva permite explorar uma ampla diversidade de soluções mesmo em cenários com restrições significativas. Os testes demonstraram desempenho consistente e boa escalabilidade.

ANÁLISE CRÍTICA

Limite Máximo de Execução:

Considerando os limites de execução definidos para 1000 tentativas, o algoritmo Generativo se mostrou capaz de resolver o problema para n até 8 de forma eficiente, com o tempo de execução dentro de parâmetros aceitáveis, a partir de $n = 12$, o tempo de execução já aumenta consideravelmente. É perceptível que esse aumento não acontece de forma linear.

Algoritmo Mais Adequado para $n \rightarrow \infty$:

Embora eficiente para tamanhos pequenos e médios de tabuleiro, enfrenta limitações devido ao crescimento exponencial do espaço de busca e ao aumento das restrições impostas pelos bloqueios no caso geral do Algoritmo Genético. Foi observado que à medida que n cresce, encontrar soluções começa a exigir bastante recurso e explorar uma quantidade muito maior de combinações, o que aumenta significativamente o tempo de execução e o número de gerações necessárias. Nesse cenário, abordagens baseadas em **busca heurística**, como o **algoritmo A***, tornam-se mais apropriadas. O A* é uma técnica eficaz por utilizar uma função heurística para guiar a busca na direção da solução, explorando apenas os estados mais promissores e reduzindo o número de verificações necessárias. Sua aplicação no problema das N-Rainhas pode ser viabilizada por meio da definição de heurísticas admissíveis que estimem o número de conflitos restantes. No entanto, para valores elevados de n , o A* pode enfrentar desafios relacionados ao consumo de memória e ao crescimento da árvore de busca.