**Trabalho 1**

**Aluno:** Gabriel Gomes Marchesan
**RGA:** 202111722011

## Resolve o problema n-rainhas com $n$ = [32, 64, 128] e com os seguintes métodos:
## A. Hill-Climbing
## B. Simulated Annealing
## C. Algoritmo Genético

### A.Hill-Climbing

**Detalhes da modelagem:** O código é composto das seguintes funções:
**configureRandomly:** Inicializa aleatoriamente o tabuleiro e o estado das rainhas como um ponto de partida para o algoritmo.
**printBoard:** Imprime o estado atual do tabuleiro.
**printState:** Imprime o estado atual das posições das rainhas.
**compareStates:** Compara dois estados para determinar se são iguais.
**calculateObjective:** Calcula o valor objetivo do estado atual, representando o número de ataques entre rainhas.
**generateBoard:** Atualiza o tabuleiro com base no estado atual.
**copyState**: Copia o conteúdo de um estado para outro.
**getNeighbour:** Encontra o vizinho com o menor valor objetivo entre os vizinhos possíveis.
**hillClimbing:** Implementa o algoritmo Hill Climbing para resolver o problema das N-Rainhas, procurando uma configuração onde as rainhas não se ataquem mutuamente.

O objetivo do algoritmo é encontrar uma configuração das rainhas onde nenhuma delas se ataquem mutuamente. Se o algoritmo encontrar uma solução, ela será impressa no final do processo. Caso contrário, o algoritmo pode chegar a uma solução local, que é uma configuração onde não é possível melhorar mais o valor objetivo, mas que pode não ser a solução global ótima para o problema.

### Resultados obtidos:

| Testes | 32(Tempo) | 32(Mínimo) | 64(Tempo) | 64(mínimo) | 128(Tempo) | 128(mínimo) |
|---|---|---|---|---|---|---|
| 1 | 2.4s | local(4) | 1m 17.5s | local(3) | 47m 15.2s | local(3) |
| 2 | 2.3s | local(3) | 1m 18.1s | local(2) | 45m 9.3s | local(3) |
| 3 | 3.1s | local(2) | 1m 27.2s | local(4) | 46m 53.1s | local(5) |
| 4 | 2.3s | local(2) | 1m 15.4s | local(2) | 45m 36.4s | local(4) |
| 5 | 2.4s | local(3) | 1m 24.6s | local(3) | 46m 27.8s | local(5) |
| **média** | **2.5s** | **local(3)** | **1m 20.5s** | **local(3)** | **46m 25.1s** | **local(4)** |

**Análise sobre o comportamento do método:** O Hill-Climbing é uma abordagem simples e eficaz para o problema das N-Rainhas, mas não é uma garantia de encontrar uma solução ótima globalmente. Ele usa um espaço de buscas (x,y), onde o eixo x são todas as possíveis combinações de Rainhas em um tabuleiro de xadrez e o eixo y é a função na qual se define a qualidade da resposta, no caso a quantidade de Rainhas que se confrontam no tabuleiro de xadrez (com tamanho definido pela quantidade de Rainhas no espaço).

## B. Simulated Annealing

**Detalhes da modelagem:** O código é composto das seguintes funções:
**threat_calculate(n):** Calcula o número total de ameaças entre rainhas em um tabuleiro de xadrez com base no número de rainhas (combinação combinatorial).
**create_board(n):** Gera um tabuleiro de xadrez aleatório com N rainhas posicionadas em linhas diferentes.
**cost(chess_board):** Calcula o custo de uma configuração do tabuleiro, ou seja, o número de pares de rainhas que se ameaçam mutuamente.
**simulated_annealing():** Implementa o algoritmo Simulated Annealing para resolver o problema das N-Rainhas. Ele gera sucessores, avalia a aceitação de novas configurações com base no custo e na temperatura e continua iterando até encontrar uma solução ou até o resfriamento atingir um limite.
**print_chess_board(board):** Imprime o tabuleiro de xadrez com as posições das rainhas.
**main():** Função principal que inicia o algoritmo Simulated Annealing e mede o tempo de execução.

O objetivo do algoritmo é encontrar uma configuração de tabuleiro onde as rainhas não se ameaçam mutuamente, inspirado pelo processo de recozimento metalúrgico (annealing) em metalurgia, o Simulated Annealing é usado para explorar o espaço de soluções com uma configuração que permite movimentos que levam a soluções piores, com uma probabilidade que diminui gradualmente ao longo do tempo (ou "temperatura"). O resultado é impresso na saída, juntamente com o tempo de execução.

## Resultados obtidos:

| Testes | 32(Tempo) | 32(Mínimo) | 64(Tempo) | 64(mínimo) | 128(Tempo) | 128(mínimo) |
|---|---|---|---|---|---|---|
| **1** | 0.06s | local(1) | 0.2s | local(56) | 0.9s | local(98) |
| **2** | 0.04s | local(27) | 0.12s | local(28) | 0.64s | local(86) |
| **3** | 0.07s | local(5) | 0.35s | local(18) | 0.6s | local(52) |
| **4** | 0.04s | local(16) | 0.2s | local(53) | 0.72s | local(22) |
| **5** | 0.08s | local(17) | 0.12s | local(58) | 0.77s | local(1) |
| **média** | 0.058s | local(13) | 0.198s | local(42) | 0.726s | local(51) |

**Análise sobre o comportamento do método:** O Simulated Annealing é uma técnica útil para resolver o problema das N-Rainhas, pois permite a exploração de diferentes configurações e a busca por soluções onde as Rainhas não se confrontam. Sua capacidade de lidar com mínimos locais e controlar a probabilidade de aceitar movimentos piores o torna uma escolha valiosa para problemas de otimização combinatorial complexos como esse.

## C. Algoritmo Genético

**Detalhes da modelagem:** O código é composto das seguintes funções:
**conflicted(state, row, col):** Verifica se colocar uma rainha na posição (row, col) entraria em conflito com qualquer outra rainha no estado atual. conflict(row1, col1, row2, col2): Verifica se colocar duas rainhas em (row1, col1) e (row2, col2) entraria em conflito.
**goal_test(state):** Verifica se todas as colunas estão preenchidas e se não há conflitos entre as rainhas no estado atual, ou seja, se a solução é válida.
**h(node):** Calcula o número de rainhas que estão se ameaçando mutuamente no estado representado por node. Quanto menor o valor retornado, melhor a configuração do tabuleiro.
**nqueen_fitness(node):** Converte a representação binária do tabuleiro node em uma representação numérica, calcula a aptidão (número de conflitos) e a retorna como uma tupla. Quanto menor a aptidão, melhor a configuração do tabuleiro.
**plot_solution(solution, N):** Plota uma solução encontrada, representando as posições das rainhas no tabuleiro de xadrez.
**creator.create("Fitness", base.Fitness, weights=(-1.0, )):** Define o tipo de aptidão para minimização, onde o objetivo é minimizar o valor de aptidão (número de conflitos).
**creator.create("Individual", list, fitness=creator.Fitness):** Define o tipo de indivíduo, que é uma lista, com uma aptidão associada.
**toolbox.register("attr_bool", random.randint, 0, 1):** Registra a função para criar valores binários (0 ou 1) que representam a presença ou ausência de rainhas em posições específicas no tabuleiro.
**toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, n=N*log_N):** Registra a função para criar um indivíduo (cromossomo) como uma lista de valores binários para todas as posições do tabuleiro.
**toolbox.register("population", tools.initRepeat, list, toolbox.individual):** Registra a função para criar uma população de indivíduos.
**toolbox.register("evaluate", nqueen_fitness):** Registra a função de avaliação que calcula a aptidão de um indivíduo.
**toolbox.register("mate", tools.cxOnePoint):** Registra a função para realizar cruzamento (crossover) entre dois indivíduos.
**toolbox.register("mutate", tools.mutFlipBit, indpb= n):** Registra a função para realizar mutação em um indivíduo com uma probabilidade de mutação de n%.
**toolbox.register("select", tools.selTournament, tournsize= n):** Registra a função para realizar seleção de pais por meio de um torneio com um tamanho de torneio de n.

**Resultados obtidos:**

| Testes | 32(Tempo) | 32(Mínimo) | 64(Tempo) | 64(mínimo) | 128(Tempo) | 128(mínimo) |
|---|---|---|---|---|---|---|
| **1** | 1m 20.2s | local(14) | 4m 30.6s | local(40) | 17m 5.8s | local(116) |
| **2** | 1m 21.4s | local(14) | 4m 36.4s | local(42) | 16m 51.1s | local(92) |
| **3** | 1m 21.5s | local(8) | 4m 36.9s | local(28) | 16m 58.7s | local(98) |
| **4** | 1m 20.4s | local(8) | 4m 36.4s | local(38) | 17m 10.1s | local(102) |
| **5** | 1m 22.1s | local(12) | 4m 36.3s | local(36) | 16m 50.9s | local(94) |
| **média** | 1m 21.s | local(11) | 4m 35.3s | local(36) | 17m 01.4s | local(100) |

**Análise sobre o comportamento do método:** Os algoritmos genéticos oferecem uma abordagem robusta e eficiente para resolver o desafiador problema das N-Rainhas, permitindo a exploração de soluções em um espaço de busca complexo e a convergência gradual para soluções cada vez melhores. A sensibilidade aos parâmetros e a necessidade de cuidadosa configuração são aspectos importantes a serem considerados ao aplicar AGs para resolver esse problema.