Inteligência Artificial - 8 Rainhas com Hill Climbing

Mesquita S. Arthur[1], Costa V. N. João[2]

[1]*8º Período de Engenharia de Computação, CEFET-MG Campus V*
Divinópolis, MG, Brasil
arthur.santana.mesquita@gmail.com
[2]*4º Período de Engenharia de Computação, CEFET-MG Campus V*
Divinópolis, MG, Brasil
joaovnevescosta@hotmail.com

Abstract. Este trabalho tem como objetivo implementar o algoritmo de *hill climbing* de forma a solucionar o problema das 8 rainhas com início em posições geradas aleatoriamente com base em uma semente previamente estipulada (42), e em seguida comparar métricas de tempo e número de reinícios entre duas variações do algoritmo, uma com movimentações laterais A atividade foi realizada em Outubro de 2025 para a disciplina de Inteligência Artifical, no curso de Engenharia de Computação do CEFET-MG - Campus V.

Keywords: Algoritmos, IA, Hill Climb, Xadrez

1 Introdução

O problema das 8 rainhas consiste em um tabuleiro de xadrez, onde devem ser posicionadas 8 rainhas, sem que nenhuma rainha seja capaz de atacar a outra, ou seja, nenhuma peça compartilhe a mesma linha, coluna ou diagonal. Para representar tal problema foram necessárias funções para gerar tabuleiros aleatórios -representados por meio de um vetor, onde a posição do termo é referente à linha do tabuleiro, e seu valor de 0 a 7 representa sua coluna - funções para calcular o número de colisões tanto do tabuleiro atual quanto dos tabuleiros possíveis a partir dos movimentos disponíveis um dado momento, e a implementação do *hill climbing* em duas de suas variações, além é claro do monitoramento para diversas medidas de desempenho nos algoritmos. A linguagem selecionada para as implementações foi Python 3.12

2 Implementação

2.1 Utilização

Para executar o programa em *python* é necessário a instalação das bibliotecas *numpy* e *matplotlib* através do seguinte comando do terminal:

```
pip install numpy matplotlib
```

E o seguinte script realiza a execução dos algoritmos a partir da raiz do projeto:

```
python trabalho2/src/QueenBoard.py
```

2.2 Queenboard.py

Foram construídas **10 funções em Python** para representar e interagir com o tabuleiro:

- **initialize_board**
  A partir da biblioteca `numpy` e da semente definida globalmente, cria valores aleatórios em novos vetores de posições de rainhas - tabuleiros - que serão utilizados para a execução do algoritmo.

- **conflicts**
  A partir de um dado tabuleiro verifica a existência de colisões entre as rainhas e às adiciona a um contador, cujo valor é retornado na forma de um valor inteiro. O cálculo das colisões ocorre de forma verifica-se primeiramente se existem rainhas na mesma linha e, em seguida, verifica se existem rainhas nas mesmas diagonais a partir da seguinte comparação:
  ```
  abs(row1 - row2) == abs(col1 - col2)
  ```
  Basicamente, duas rainhas estão na mesma diagonal (principal ou secundária) se a diferença em módulo de suas linhas é igual a diferença em módulo de suas colunas.
- **neighbors**
  A partir de um dado tabuleiro verifica quais movimentações de rainha são válidas em um dado momento - ou seja, quais os vizinhos do estado atual no *hill climbing*
- **apply_move**
  A partir de uma tupla de coluna e linha - rainha a mover e para qual linha movê-la - gera-se um novo tabuleiro com o movimento aplicado.
- **generate_dict_conflicts**
  A partir de um dado tabuleiro cálcula o número de conflitos de cada um de seus vizinhos, os salva em um dicionário, onde a chave é o movimento que gera aquele vizinho e o valor é o número de conflitos. Em seguida este dicionário é ordenado, de forma que os vizinhos com menor número de conflitos são colocados no início do dicionário.
- **best_move**
  A partir de um dicionário de conflitos, retorna-se o primeiro termo do dicionário, que sempre será o melhor vizinho (com menor número de conflitos).
- **hill_climb_with_random**
  A partir de um dado tabuleiro, executa um loop que é encerrado apenas quando o número de conflitos é reduzido a zero. Em cada execução deste loop há uma chance de 5% de o tabuleiro atual ser descartado e um novo ser gerado com `initialize_board`, incrementando o contador de reinícios. Em seguida é gerado o dicionário de conflitos através de `generate_dict_conflicts` e o primeiro (e com menos conflitos) vizinho do dicionário é selecionado e aplicado, gerando um novo tabuleiro e reiniciando o loop.
- **hill_climb_with_lateral**
  A partir de um dado tabuleiro e um limite de saltos laterais (10), executa um loop que é encerrado apenas quando o número de conflitos é reduzido a zero. Em cada execução deste loop a função `next_move_with_lateral` é executada para decidir o próximo passo e este é aplicado ao tabuleiro atual, que é atualizado. Porém, caso `next_move_with_lateral` retorne o movimento inválido (8, 8) - condição que é forçada quando não encontrou-se movimento válido dentro do limite de saltos laterais (10) - o tabuleiro atual é descartado e um novo é gerado com `initialize_board`, incrementando o contador de reinícios. Em seguida o loop é executado novamente.
- **next_move_with_lateral**
  A partir de um dado tabuleiro e um limite de saltos laterais (10), gera o dicionário de conflitos dos vizinhos por meio de `generate_dict_conflicts`, e o número de conflitos do tabuleiro atual por meio de `conflicts`, salvando-se o melhor (primeiro) termo do dicionário para comparação. Depois os vizinhos com menor número de conflitos são salvos em um vetor, com o vetor sendo esvaziado toda vez que um novo mínimo de conflitos é encontrado dentre os vizinhos. Por fim, caso o mínimo de conflitos dentre os vizinhos é menor que o número atual de conflitos este é escolhido, o contador de testes é reiniciado e o movimento e retornado pela função. Caso contrário, continua-se a percorrer o vetor de melhores vizinhos e incrementar o contador de testes até que um novo estado válido seja encontrado, o limite de saltos laterais (10) seja atingido ou ainda o vetor de melhores vizinhos se esvazie. Neste caso o movimento inválido (8, 8) é forçadamente retornado pela função.
- **main**
  É a função principal do programa, faz a execução dos dois algoritmos de *Hill Climbing* e compara-os em duas condições: Para uma solução e para as 92 soluções do problema das 8 rainhas.
  Além disso, essa função é responsável por medir as métricas de desempenho e gerar os gráficos de barra para essas métricas. Desta função, as outras são chamadas automaticamente.

2.3 GraphGenerator.py

Script usado para gerar os gráficos de barra usados neste relatório usando `matplotlib`. Apresenta uma única função:

- **generate_bar_chart**
  Recebe como parâmetros: Um dicionário, sendo, geralmente, as chaves como os métodos (movimentação local e reinício aleatório) e os valores são usados no eixo y do gráfico; O título; Nome do eixo x e y; Nome do arquivo e um dicionário de cores (opcional), para as cores das barras.
  Define o tamanho da figura, implementa barra a barra com etiquetas de valor a partir do dicionário (em todos os casos foram usadas apenas duas barras), define todos os nomes e salva a figura em `trabalho2/output`.

3 Analise de Resultados

Durante a execução do código, foram aferidas as métricas de tempo de execução e número de reinícios tanto para gerar uma única solução, quanto para encontrar todas as 92 possíveis soluções para o problema. E com base na geração das 92 soluções também foi aferida a taxa de sucesso e média do número de movimentações até chegar a um resultado. Assim os valores foram comparados entre as duas variantes do *hill climbing*, gerando a tabela a seguir:

Algoritmo	Tempo (ms)	Número de Reinícios	Média de Movimentações	Taxa de Sucesso
Movimento Lateral (1 solução)	5.00	3	3.67	
Reinícios Aleatórios (1 solução)	24.00	8	16.25	
Movimento Lateral (92 soluções)	1469.33	139	4.43	65.71%
Reinícios Aleatórios (92 soluções)	4993.31	216	9.89	42.40%

Table 1. Comparação entre as variações do algoritmos *hill climbing* para solução única e todas as soluções.

Verifica-se que o algoritmo de *Hill Climbing com Movimentação Lateral Limitada* obteve melhor desempenho em todas as métricas. Com os gráficos gerado explicitando sua superioridade sobre o *Hill Climbing com Reinícios Aleatórios* para os valores selecionados de limite.

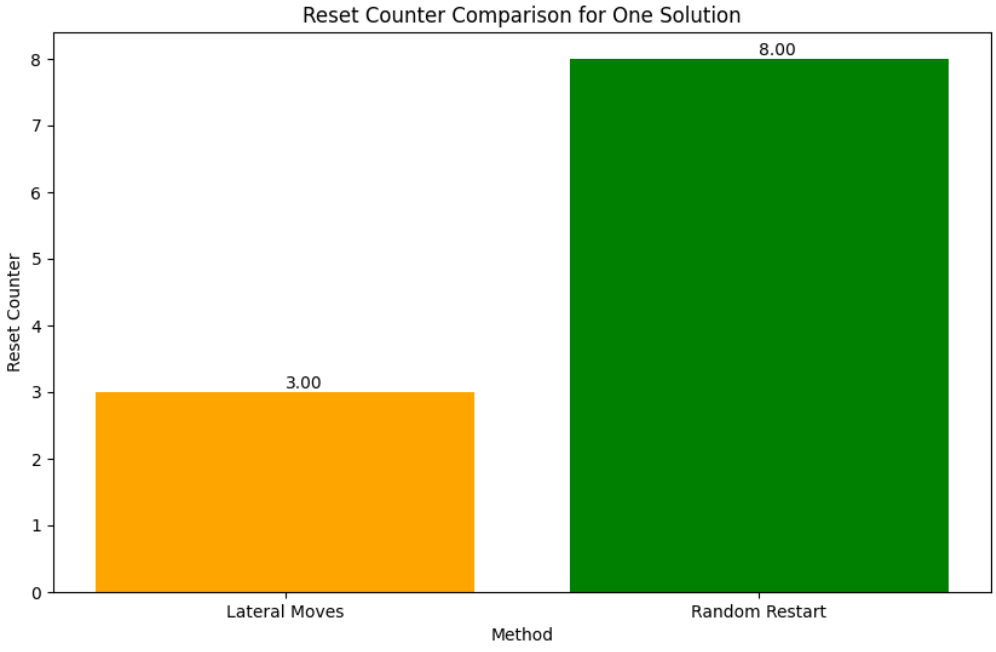

Figure 1. Gráfico de número de reinícios para geração solução única. Menor é melhor

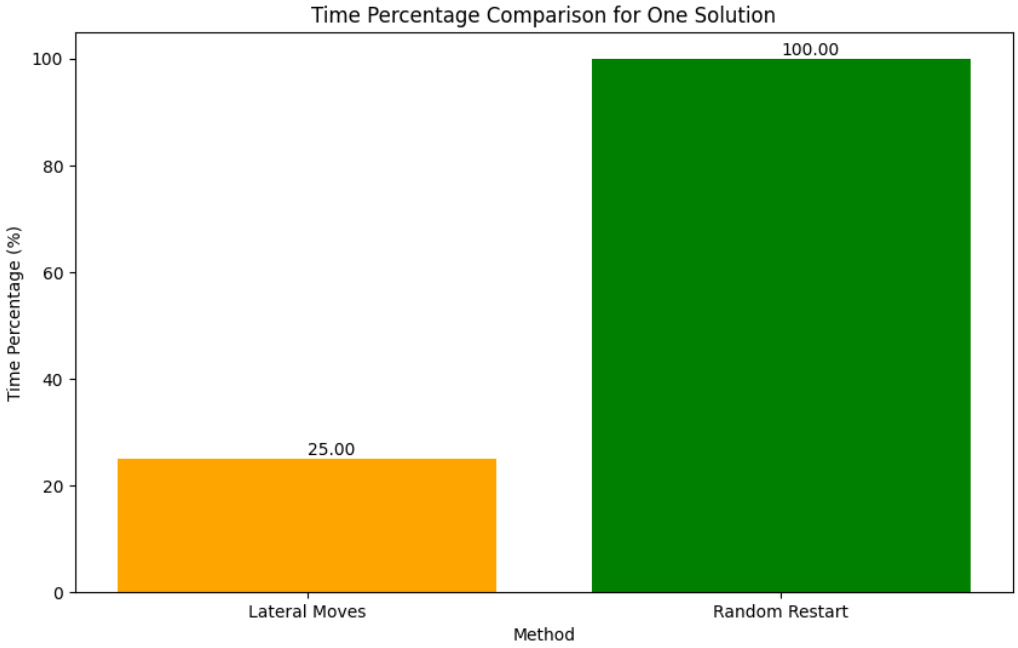

Figure 2. Gráfico da porcentagem do tempo de execução para geração de solução única.

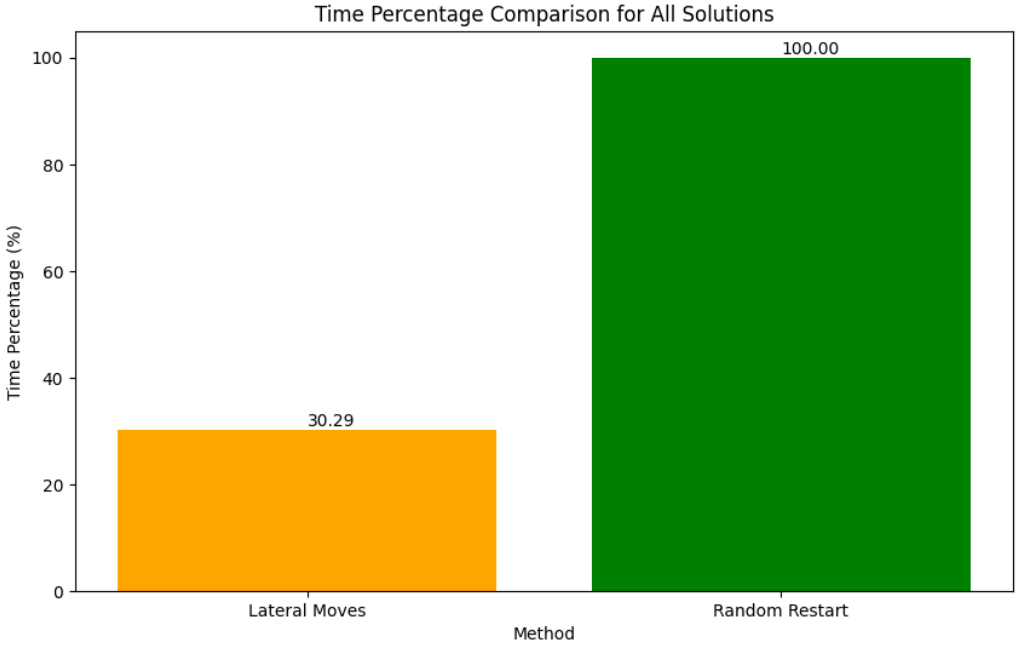

Figure 3. Gráfico da porcentagem do tempo de execução para geração de todas as soluções.

Percebe-se que o algoritmo de *Hill Climbing com Reinícios Aleatórios*, que demorava 4 vezes mais em sua execução que o *Movimentação Lateral Limitada* para geração de solução única teve uma melhora de desempenho quando executado para geração de todas as soluções, sendo apenas 3.25 vezes mais lento nesta métrica.

Outra métrica de interesse é que o número total de movimentações - número de reinícios multiplicado pela média de movimentações - foi 10 vezes maior para *Reinícios Aleatórios* que para *Movimentação Lateral Limitada* na geração de solução única.

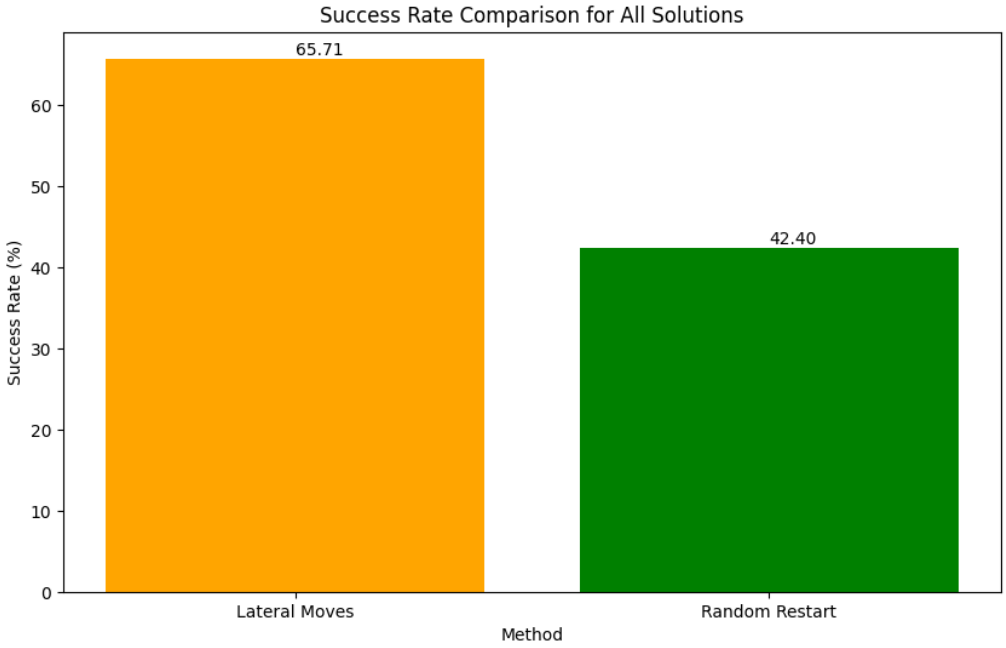

Figure 4. Gráfico de taxa de sucesso por iteração para geração de todas as soluções. Maior é melhor

Por fim, é importante pontuar que a taxa de sucesso do algoritmo de *Movimentação Lateral Limitada* é 54% melhor que *Reinícios Aleatórios*, o que garantiu o número substancialmente menor de reinícios deste algoritmo.

4 Conclusão

Conclui-se portanto, a importância de uma boa escolha de algoritmo para cada problema. O problema das 8 rainhas é estruturado de forma que um grande número de reinícios aleatórios não se faz tão efetivo para evitar picos locais. Pode-se porém, inferir que uma vantagem surgiria quando em um problema com platôs maiores, que podem gerar saltos inefetivos, ou ainda, pode-se pensar que uma escolha diferente de limite de saltos laterais poderia levar a uma perda de desempenho que justificasse o uso de saltos aleatórios.

Assim não pode-se ignorar a possibilidade de utilização de outras estratégias de hill climbing - como por exemplo *Simulated Annealing Hill Climbing* que usa de saltos aleatórios para seleção de um pivô, com a distância dos saltos diminuindo em torno do pivô ao longo do tempo de execução - tanto como estratégia de solução do problema das 8 - ou N - rainhas, quanto para outros problemas que exigem Inteligência Artificial.

5 Créditos

- **João Vitor Neves Costa:** Implementação `initialize_board`, `conflicts`, `neighbors`, `main`. escrita seções *1, 2.5, 2.6, 4, 6*
- **Arthur Santana de Mesquita:** Implementação `hill_climb_with_random`, `hill_climb_with_lateral`, `best_conflict`, `generate_dict_conflicts`. análise de desempenho, escrita seções *2.1, 2.3, 2.4, 3*

6 Referências

References

[1] Ai — search algorithms — hill climbing, 06 2023. URL https://www.codecademy.com/resources/docs/ai/search-algorithms/hill-climbing.

[2] Alvaro Leandro Cavalcante Carneiro. Algoritmos de otimização: Hill climbing e simulated annealing, 06 2020. URL https://medium.com/data-hackers/algoritmos-de-otimiza%C3%A7%C3%A3o-hill-climbing-e-simulated-annealing-3803061f66f0.

[3] Federer. Understanding generators in python, 11 2009. URL https://stackoverflow.com/questions/1756096/understanding-generators-in-python.
[4] Francisco Iacobelli. hillclimbing 8 queens, 06 2015. URL https://www.youtube.com/watch?v=vEpPMIiTSDI.
[5] Parfi. How to write text above the bars on a bar plot (python)?, 11 2016. URL https://stackoverflow.com/questions/40489821/how-to-write-text-above-the-bars-on-a-bar-plot-python.
[6] Stuart J Russell and Peter Norvig. *Inteligência Artificial - Uma Abordagem Moderna*. GEN LTC, 08 2022.
[7] Bex Tuychiev. Implementing the hill climbing algorithm for ai in python, 02 2025. URL https://www.datacamp.com/tutorial/hill-climbing-algorithm-for-ai-in-python.