Centro Federal de Educação Tecnológica de Minas Gerais
Engenharia de Computação
Inteligência Artificial

Trabalho 02:
8 Rainhas com *Hill Climbing*

Guilherme Alvarenga de Azevedo
Maria Eduarda Teixeira Souza

Divinópolis - MG
Novembro de 2025

Sumário

Problema **3**
- **Introdução** 3
- **Descrição** 3
  - **Objetivo** 3
  - **O Tabuleiro** 3
- **Modelagem** 4
  - **Representação como Lista** 4
  - **Estruturas de Dados** 5
  - **Tamanho e Complexidade** 5
- **Algoritmos** 5
  - ***Hill Climbing* com Movimentos Laterais** 6
  - ***Hill Climbing* com Reinícios Aleatórios** 6
  - ***Simulated Annealing*** 6
- **Critério de Análise e Avaliação** 7
- **Organização e Execução do Projeto** 8
  - **Organização do diretório** 8
  - **Visualização** 9
  - **Variantes e rótulos nos resultados** 9
  - **Instalação e Execução** 10

Hill Climbing **12**

Hill Climbing* com *Sideways Moves **13**
- *Inicialização e configuração* 13
- *Geração e avaliação de vizinhos* 13
- *Movimentação e controle de sideways* 13
- *Memória e histórico de estados* 14
- *Limite de memória para visited_boards* 15
- **Análise Teórica** 15
- **Soluções Encontradas** 15
- **Comparação de Desempenho Real** 16
- **Comparação gráfica de desempenho** 16

Hill Climbing* com *Random Restarts **18**
- *Inicialização e configuração* 18
- *Estrutura do laço de reinícios* 18

Geração de tabuleiros aleatórios . . . 19
Rastreamento e histórico . . . 20
Análise Teórica . . . 20
Soluções Encontradas . . . 21
Comparação de Desempenho Real . . . 21
Comparação gráfica de desempenho . . . 22

Simulated Annealing . . . **23**
Representação do estado e avaliação . . . 23
Aceitação probabilística de piores estados . . . 23
Função de resfriamento (cooling schedule) . . . 24
Rastreamento de histórico e estados . . . 24
Análise Teórica . . . 25
Soluções Encontradas . . . 25
Comparação de Desempenho Real . . . 25
Comparação Gráfica de Desempenho . . . 26

Comparação Geral dos Algoritmos . . . **28**
Análise de Tempo de Execução . . . 28
Número Médio de Conflitos . . . 29
Taxa de Sucesso . . . 29
Número Médio de Passos e Memória . . . 30
Memória Corrente Média . . . 30
Memória Máxima . . . 31

Conclusão . . . **32**

Créditos e Declaração de Autoria . . . **33**
Autores e Papéis . . . 33
Uso de IA . . . 34

Referências - Recursos Externos . . . **35**

Declaração . . . **36**

Problema

Introdução

Este trabalho aborda a implementação e análise do problema das 8 Rainhas, um clássico da computação e da inteligência artificial, utilizando métodos de busca local. A proposta consiste em empregar o algoritmo *Hill Climbing* e suas variações, bem como o algoritmo *Simulated Annealing*, para encontrar uma configuração válida das rainhas no tabuleiro de xadrez, de forma que nenhuma delas entre em conflito. O estudo busca compreender o comportamento, desempenho e limitações de cada abordagem na resolução de problemas de otimização com múltiplos máximos locais.

Descrição

O problema das 8 Rainhas consiste em posicionar oito rainhas em um tabuleiro de xadrez 8×8 de forma que nenhuma delas ataque outra. Em termos práticos, isso significa garantir que não existam duas rainhas na mesma linha, coluna ou diagonal. Embora o problema possua diversas soluções válidas, sua complexidade cresce rapidamente conforme o número de rainhas aumenta, o que o torna um caso de estudo interessante para técnicas de busca heurística e otimização local.

Objetivo

O objetivo deste trabalho é implementar e comparar duas variações do algoritmo *Hill Climbing* — uma com movimentos laterais limitados e outra com reinícios aleatórios (*Random-Restart*) — além de aplicar o algoritmo *Simulated Annealing* como contraste. Por meio dessa comparação, busca-se analisar a eficiência, o tempo de convergência e a taxa de sucesso de cada método na obtenção de uma solução livre de conflitos.

O Tabuleiro

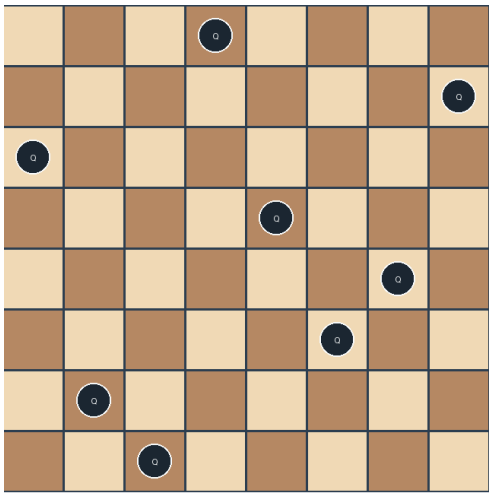
O tabuleiro do problema das 8 Rainhas é composto por 64 casas dispostas em 8 linhas e 8 colunas, onde cada posição pode conter ou não uma rainha. Para fins computacionais, a representação do tabuleiro foi simplificada por meio de um vetor unidimensional de tamanho 8, em que o índice do vetor corresponde à coluna e o valor armazenado representa a linha em que a rainha se encontra naquela coluna. Veja um exemplo:

```
random_board = [2, 6, 7, 0, 3, 5, 4, 1]
```

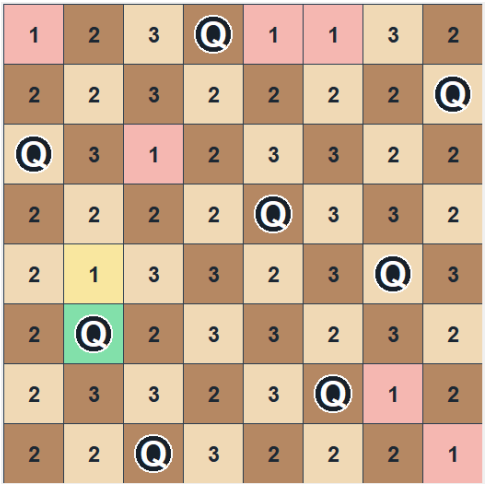
Esse vetor é gerado aleatoriamente a cada vez que o programa é inicializado. A partir desse vetor, é possível reconstruir a matriz correspondente ao tabuleiro, calcular o número de conflitos entre rainhas (pares que se atacam) e aplicar os algoritmos de busca para encontrar uma configuração sem conflitos.

```
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . . Q .
. . . . . Q . .
. . Q . . . . .
. . . Q . . . .
```

A Figura 1(a) apresenta um exemplo de disposição aleatória inicial, enquanto a Figura 1(b) mostra o tabuleiro resolvido após a execução do algoritmo.



((a)) Exemplo de Tabuleiro com as 8 Rainhas posicionadas aleatoriamente.



((b)) Exemplo de Tabuleiro com as 8 Rainhas posicionadas sem conflitos.

Figura 1 – Figuras do tabuleiro

Modelagem

A modelagem do problema das 8 Rainhas foi desenvolvida de forma a garantir uma implementação modular e flexível, permitindo testar diferentes variações dos algoritmos de busca local. Mantendo a representação proposta anteriormente, o sistema foi estruturado em funções e classes que lidam separadamente com a geração de tabuleiros, cálculo de conflitos, geração de vizinhos e execução dos métodos heurísticos. Dessa forma, cada algoritmo pôde ser avaliado isoladamente, utilizando a mesma base de representação e avaliação de estados.

Representação como Lista

O tabuleiro é representado como uma lista unidimensional de tamanho 8, em que cada índice corresponde a uma coluna e o valor armazenado indica a linha onde a rainha está posicionada. Essa estrutura simplifica tanto o armazenamento quanto a manipulação dos estados. Por exemplo, a lista `[2, 6, 7, 0, 3, 5, 4, 1]` representa um tabuleiro onde a rainha da primeira coluna ocupa a linha 2, a da segunda coluna ocupa a linha 6, e assim por diante.

Essa representação permite gerar facilmente novos estados (vizinhos) ao mover uma rainha em sua respectiva coluna. Além disso, a verificação de conflitos é realizada por meio de comparações entre pares de rainhas, verificando se elas compartilham a mesma linha ou se estão em diagonais com a mesma diferença entre linha e coluna.

Estruturas de Dados

A principal estrutura utilizada na implementação é uma lista de inteiros, onde cada índice representa uma coluna do tabuleiro e o valor armazenado indica a linha da rainha naquela coluna. Essa forma de representação é compacta, eficiente e permite avaliar rapidamente o número de conflitos entre as rainhas.
Além do vetor principal, são utilizadas listas auxiliares para armazenar os estados vizinhos gerados ao mover cada rainha em sua respectiva coluna. Em algumas variações, estruturas adicionais foram aplicadas: uma pequena lista de controle (*Tabu List*) para evitar repetições de estados no *Hill Climbing* com movimentos laterais e contadores de reinicializações no método com *Random Restarts*. No caso do *Simulated Annealing*, é empregada uma variável de temperatura que decresce conforme a função de resfriamento definida.
Essas estruturas são manipuladas por funções modulares, como `compute_conflicts()`, `generate_neighbors()` e `get_best_neighbor()`, que centralizam as operações comuns a todas as variações e contribuem para uma implementação mais organizada e reutilizável.

Tamanho e Complexidade

O espaço de busca total do problema é de $8^8 = 16.777.216$ possíveis configurações, já que cada uma das oito rainhas pode ocupar qualquer uma das oito linhas disponíveis. Embora esse número seja relativamente grande, a busca local não explora o espaço de forma exaustiva, focando apenas nos estados vizinhos a cada iteração.
O cálculo do número de conflitos para um estado possui complexidade $O(n^2)$, pois envolve a comparação entre todos os pares de rainhas. A geração de vizinhos, por sua vez, possui complexidade $O(n^2)$ também, uma vez que, para cada coluna, são testadas as demais posições possíveis. Apesar disso, o custo computacional é baixo o suficiente para permitir execuções rápidas em todas as variações implementadas.

Algoritmos

A etapa de desenvolvimento incluiu três variações do método de *Hill Climbing*, cada uma explorando uma abordagem diferente para evitar armadilhas de máximos locais e platôs. Todas as implementações compartilham a mesma função de avaliação, que calcula o número de pares de rainhas em conflito, e utilizam a geração de vizinhos por movimentação vertical de uma única rainha em sua respectiva coluna. As diferenças principais estão no controle de movimentos, critérios de aceitação e estratégia de reinicialização.

Hill Climbing com Movimentos Laterais

Nesta variação, o algoritmo busca sempre o vizinho de melhor *fitness* (menor número de conflitos), movendo a rainha que leva o tabuleiro a uma configuração mais estável. Caso o melhor vizinho possua o mesmo valor de avaliação que o estado atual, o algoritmo permite um número limitado de movimentos laterais — definidos pelo parâmetro `max_sideways_moves`. Essa estratégia busca evitar que o processo fique preso em platôs, regiões do espaço de busca onde múltiplos estados possuem o mesmo número de conflitos. O controle desses movimentos é feito por um contador, reiniciado sempre que ocorre uma melhora no *fitness*.
Além disso, foi implementada uma lista de controle similar a uma *Tabu List*, que armazena até os oito últimos estados visitados, prevenindo a repetição cíclica de configurações. Quando o limite de movimentos laterais é atingido sem melhora, o processo é encerrado.
Essa abordagem apresentou desempenho consistente, com bom equilíbrio entre tempo de execução e taxa de sucesso, embora ainda possa estagnar em platôs extensos.

Hill Climbing com Reinícios Aleatórios

O *Hill Climbing* com reinícios aleatórios (Random-Restart) foi implementado para superar o problema de convergência prematura a máximos locais. Nesse método, o algoritmo executa múltiplas buscas independentes, cada uma iniciando a partir de um tabuleiro gerado aleatoriamente.
Cada tentativa segue a lógica clássica do *Hill Climbing* — buscando o melhor vizinho até não haver mais melhora — e, ao final, o resultado é comparado ao melhor obtido até então. Caso nenhuma solução sem conflitos seja encontrada, o algoritmo reinicia até atingir o limite definido por `max_restarts`.
A função `hill_climbing_with_random_restarts()` controla todo o processo, armazenando o número de reinicializações e o histórico de conflitos da melhor execução. Essa abordagem demonstrou maior taxa de sucesso na obtenção de soluções completas, ainda que o tempo total de execução possa aumentar devido ao custo dos múltiplos reinícios.

Simulated Annealing

Como forma de contraste, foi implementada também a variação baseada no algoritmo de *Simulated Annealing*. Diferente das abordagens puramente determinísticas do *Hill Climbing*, o método utiliza uma estratégia probabilística para escapar de máximos locais, aceitando temporariamente estados piores com base em uma probabilidade que depende da diferença de conflitos e da temperatura atual do sistema.
A temperatura inicial é definida pelo parâmetro `temperature`, e o decaimento ocorre conforme a função de resfriamento selecionada: linear (`cooling_func = 1`) ou exponencial (`cooling_func = 2`). A cada iteração, um vizinho aleatório é escolhido, e a decisão de substituição é feita conforme o critério de Metropolis:

$$P(\text{aceitação}) = e^{\frac{\Delta E}{T}}$$

em que ΔE é a variação no número de conflitos e T é a temperatura atual.
Conforme a temperatura diminui, a probabilidade de aceitar movimentos piores se torna cada vez menor, fazendo com que o algoritmo gradualmente se comporte como um *Hill Climbing* tradicional. Essa estratégia se mostrou mais robusta contra platôs extensos e máximos locais, porém com maior variabilidade nos tempos de execução devido ao fator aleatório de aceitação.

Critério de Análise e Avaliação

A avaliação do desempenho dos algoritmos de busca local aplicados ao problema das 8 Rainhas foi conduzida a partir de um conjunto de métricas que quantificam tanto a eficiência computacional quanto a qualidade da solução obtida. Cada algoritmo foi executado 100 vezes, sob as mesmas condições iniciais — tabuleiros aleatórios pré-gerados e controlados — de modo a garantir a comparabilidade entre os resultados e a reprodutibilidade experimental.
Entre as métricas gerais coletadas, destacam-se o *tempo médio de execução* (*avg time*), medido em milissegundos, que expressa a rapidez da convergência do algoritmo; o *consumo médio de memória* (*avg memory*), obtido por meio do monitoramento do uso de memória residente (RSS); e os valores de *memória atual média* (*avg current*) e *pico de memória* (*avg peak*), ambos medidos com *tracemalloc*, que indicam, respectivamente, a alocação final e o maior uso de memória durante o processo.
Além disso, são observados os *passos médios* (*avg steps*) até o término da execução, a *taxa de sucesso* (*success rate*) — proporção de execuções que encontraram uma solução sem conflitos — e o número de *conflitos médios* (*avg conflicts*), indicador direto da qualidade final da configuração.
Cada variação do algoritmo inclui, contudo, parâmetros específicos que influenciam sua análise. No *Hill Climbing* com movimentos laterais, por exemplo, o limite de movimentos permitidos é registrado como *sideways limit*, representando a tolerância a platôs de mesmo valor de avaliação. Já no *Random-Restart Hill Climbing*, são adicionadas as métricas *avg restarts* e *max moves*, que expressam, respectivamente, o número médio de reinicializações e o limite de movimentos por tentativa. Por fim, o *Simulated Annealing* possui métricas próprias de controle térmico, como a *temperatura inicial* e o número máximo de passos, além da distinção entre os esquemas de resfriamento linear e exponencial.
Essas medições permitem não apenas comparar o desempenho entre algoritmos, mas também compreender como diferentes estratégias de exploração — reinicializações, movimentos laterais ou aceitação probabilística — impactam o tempo de convergência, o uso de recursos e a capacidade de escapar de mínimos locais.

Organização e Execução do Projeto

O repositório do *Github* [1] contém as implementações dos algoritmos de busca local e a infraestrutura para comparação numérica e visual, medição de recursos e geração de gráficos. As variantes implementadas incluem:

- Subida da encosta (Hill Climbing) com movimentos laterais (limites de 10 e 100);
- Reinícios aleatórios (Random Restarts) com Subida da Encosta, nas versões com e sem movimentos laterais (limite de 20 movimentos por reinício, até 100 reinícios);
- Recozimento simulado (Simulated Annealing) com *cooling* linear e exponencial (1000 passos por padrão).

Todas as comparações são executadas por padrão em 100 rodadas por variante, compartilham o mesmo conjunto de 100 tabuleiros iniciais e salvam métricas e gráficos automaticamente.

Organização do diretório

A estrutura do Trabalho 2 separa dados, código-fonte e utilitários de execução:

- **data/** — resultados gerados:
  - **output/metrics/** — arquivo JSON com as métricas agregadas: *metrics_hill_climbing.json*;
  - **output/graphics/** — contém todos os gráficos gerados a partir das métricas dos algoritmos, organizados em subpastas:
    - **comparisons/** — gráficos que comparam diretamente pares de algoritmos: *Restarts-Comparison.png* (comparação entre *RandomRestartsSideways* e *RandomRestartsHill*), *SA-Comparison.png* (comparação entre *SimulatedAnnealingLinear* e *SimulatedAnnealingExponential*) e *Sideways-Comparison.png* (comparação entre *Sideways-10* e *Sideways-100*).
    - **general/** — gráficos de comparação geral entre todos os algoritmos, incluindo métricas como tempo médio, conflitos médios, passos médios, memória atual e máxima, e taxa de sucesso.
    - **progress/** — gráficos que representam a evolução do estado da solução ao longo das iterações para cada tipo de simulação, totalizando seis gráficos, permitindo visualizar o progresso das soluções em cada algoritmo.
  - **output/visualization** — gifs gerados para visualização.
- **src/** — código-fonte do projeto.
  - **core/** — representação do problema das 8 rainhas (tabuleiro, conflitos, vizinhança).

  - **local_search/** — algoritmos de busca local: *hill_climbing.py*, *sideways_moves.py*, *random_restarts.py* e *simulated_annealing.py*.
  - **comparisons/** — comparação e gráficos: *compare_hill_climbing.py* (gera as métricas e chama a plotagem) e *hill_climbing_plots.py* (gráficos).
  - **tools/** — scripts de execução: *main.py* (CLI) e *run_gui.py* (interface gráfica), além de *measure_time_memory.py* para mensurar tempo/memória.
  - **visualization/** — utilitário para GIFs de estados: *queen_gif.py*.

- Arquivos de configuração e documentação, como `pyproject.toml`, `requirements.txt`, `README.md` e `relatorio.pdf`.

Visualização

O projeto oferece uma interface gráfica (GUI) para acompanhar, passo a passo, o comportamento dos algoritmos e a movimentação das rainhas no tabuleiro de acordo com o algoritmo selecionado. A GUI exibe o tabuleiro, destaca os melhores candidatos por passo e mostra o histórico de conflitos. Além disso, permite ajustar parâmetros (limite de movimentos laterais, reinícios, temperatura, função de *cooling*, passos), executa comparações entre variantes e salva métricas e gráficos automaticamente.
Além da GUI, o projeto pode ser executado via *main.py* no terminal. A visualização da evolução dos estados pode ser salva como GIF através do utilitário *visualization/queen_gif.py* (a GUI inclui um botão "Save GIF..." para exportação). As métricas agregadas são registradas em *data/output/metrics/metrics_hill_climbing.json* e os gráficos comparativos são salvos em *data/output/graphics/*.

Variantes e rótulos nos resultados

Foram implementadas e comparadas seis variantes principais do problema das N-Rainhas, cada uma com parâmetros e métricas específicas. As duas primeiras correspondem ao *Hill Climbing* com movimentos laterais limitados a 10 e 100 tentativas, rotuladas respectivamente como *Sideways (10)* e *Sideways (100)*, e identificadas nas métricas pelos prefixos *Sideways-10* e *Sideways-100*.
As duas variantes seguintes incorporam reinicializações aleatórias: a primeira combina o método de reinícios com movimentos laterais (*Restarts + Sideways*, prefixo *RandomRestartsSideways*), permitindo até 20 movimentos e 100 reinicializações; a segunda utiliza o mesmo esquema, mas sem permitir movimentos laterais (*Restarts + Hill*, prefixo *RandomRestartsHill*).
Por fim, foram avaliadas duas versões do *Simulated Annealing*, diferenciadas pela função de resfriamento: uma linear (*SA Linear*, prefixo *SimulatedAnnealingLinear*) e outra exponencial (*SA Exp*, prefixo *SimulatedAnnealingExponential*), ambas com temperatura inicial igual a 400 e limite de 1000 passos. Todas as variantes foram executadas sobre as mesmas 100 configurações

iniciais de tabuleiro, pré-geradas de forma determinística, e analisadas com base em chaves de métrica no formato "*<Algoritmo> <Critério>*" — como, por exemplo, "*Sideways-10 avg time (ms)*" ou "*SimulatedAnnealingLog avg conflicts*". Para os algoritmos estocásticos, o gerador pseudoaleatório é inicializado a cada execução, assegurando reprodutibilidade e diversidade nas amostras.

Instalação e Execução

Para utilizar o projeto, siga os passos abaixo:

1. Clone o repositório e entre na pasta do projeto:

```
git clone https://github.com/dudatsouza/ia-trabalhos.git
cd ia-trabalhos/trabalho2
```

2. Crie e ative um ambiente virtual (recomendado) - garanta que já possui o Python, no mínimo na versão 3.11.9:

```
python3 -m venv venv
source venv/bin/activate   # Linux/macOS
venv\Scripts\activate      # Windows
```

3. Instale as dependências com Poetry - garanta que possui o Poetry instalado:

```
poetry install
```

4. Alternativamente, instale as dependências com pip:

```
pip install -r requirements.txt
```

5. Caso o *tkinter* não esteja instalado (necessário para GUI):

```
sudo apt-get install python3-tk   # Linux
```

   Windows e macOS já incluem o *tkinter*.

6. Execute o programa:

   Com GUI (<modulo> = *run_gui*) ou sem GUI (<modulo> = *main*):

   - Linux/macOS:

```
PYTHONPATH='src' python3 -m tools.<modulo>
```

     ou

```
poetry run python src/tools/<modulo>.py
```

   - Windows:

```
set PYTHONPATH=src && python -m tools.<modulo>
```

ou

```
poetry run python src/tools/<modulo>.py
```

O projeto depende das bibliotecas *matplotlib* e *tkinter*. Todas as informações sobre as dependências também estão presentes no arquivo *requirements.txt*.

Hill Climbing

O algoritmo *Hill Climbing* é uma técnica clássica de busca local utilizada para resolver problemas de otimização, como o das N-Rainhas. Sua ideia central consiste em partir de uma configuração inicial e realizar sucessivas melhorias incrementais até atingir um estado que não possua vizinhos melhores — ou seja, um máximo local. No contexto das N-Rainhas, cada estado representa a disposição das rainhas no tabuleiro, e a função de avaliação (*fitness*) mede o número de pares de rainhas que se atacam mutuamente; portanto, o objetivo do algoritmo é minimizar esse valor até chegar a zero, indicando uma solução válida.

A implementação emprega uma representação de estado compacta, em que o tabuleiro é descrito por uma lista de inteiros — cada índice corresponde a uma coluna, e o valor armazenado indica a linha em que a rainha está posicionada. A cada iteração, o algoritmo gera todos os vizinhos possíveis, movendo uma única rainha verticalmente dentro de sua coluna. O vizinho com menor número de conflitos é então selecionado como o próximo estado. Caso nenhum vizinho melhore a avaliação atual, o algoritmo encerra, caracterizando a convergência a um máximo local.

Para contornar esse comportamento estagnante, foram exploradas variantes que introduzem movimentos laterais (*sideways moves*) — permitindo aceitar estados com a mesma pontuação por um número limitado de vezes — e reinicializações aleatórias (*random restarts*), que reiniciam a busca a partir de novas configurações quando não há progresso. Essas estratégias tornam o método mais robusto diante de platôs e picos locais, aumentando a taxa de sucesso na obtenção de soluções completas sem comprometer a simplicidade do *Hill Climbing*.

Hill Climbing com *Sideways Moves*

O algoritmo *Hill Climbing* com movimentos laterais (*Sideways Moves*) é uma variação do método clássico de subida de encosta que busca amenizar o problema de estagnação em platôs — regiões do espaço de busca onde múltiplos estados possuem o mesmo valor de avaliação. Diferentemente do *Hill Climbing* tradicional, que aceita apenas movimentos que melhoram o valor da função objetivo, esta versão permite transições para estados de igual qualidade por um número limitado de vezes. Essa estratégia possibilita que o algoritmo "escape" de áreas planas do espaço de busca e continue progredindo em direção a um mínimo global.
A implementação da variação com movimentos laterais mantém a estrutura básica do *Hill Climbing*, mas adiciona elementos-chave para controlar e registrar os movimentos laterais, garantindo tanto a eficiência quanto a reprodutibilidade dos experimentos.

Inicialização e configuração

O algoritmo permite que um tabuleiro inicial seja fornecido; caso contrário, ele gera uma configuração aleatória. Isso é útil para experimentos comparativos, pois garante que diferentes variantes possam ser testadas sobre o mesmo ponto de partida. Além disso, são inicializados:

- **Contador de movimentos laterais** (`sideways_moves`) — rastreia quantos movimentos laterais consecutivos foram realizados.
- **Histórico de conflitos** (`history`) — armazena o número de conflitos por iteração.
- **Lista de estados visitados** (`visited_boards`) — evita ciclos curtos.

Geração e avaliação de vizinhos

Para cada estado atual, o algoritmo gera todos os vizinhos possíveis (movimentos verticais das rainhas dentro de suas colunas) utilizando a função `problem.neighbors()`. Cada vizinho é avaliado por `problem.fitness()` e comparado com o estado atual:

```
1 if tuple(candidate) in visited_boards:
2     continue
```

Listing 1 – Evita ciclos curtos verificando estados já visitados.

Essa verificação impede que o algoritmo retorne imediatamente a estados recentemente visitados, promovendo uma exploração mais eficiente dos platôs.

Movimentação e controle de sideways

O núcleo do algoritmo decide se o próximo estado é aceito:

- **Melhor que o atual**: o estado é adotado e o contador de movimentos laterais é reiniciado.
- **Igual ao atual** (movimento lateral): o estado é adotado e o contador é incrementado. O parâmetro `sideways_limit` define o número máximo de movimentos laterais consecutivos permitidos.
- **Pior que o atual**: a busca termina imediatamente.

Trecho simplificado ilustrativo:

```
sideways_moves = 0
while sideways_moves < sideways_limit:
next_state = get_best_neighbor(current_state)
next_fitness = compute_fitness(next_state)

```
if next_fitness < current_fitness:
    current_state = next_state
    current_fitness = next_fitness
    sideways_moves = 0 # reinicia o contador
elif next_fitness == current_fitness:
    sideways_moves += 1 # movimento lateral
    current_state = next_state
else:
    break
```
```

Listing 2 – Controle de movimentos laterais no Hill Climbing.

Memória e histórico de estados

Além de evitar ciclos curtos, o algoritmo mantém:

- `history` — registra o número de conflitos a cada iteração para análise posterior.
- `states` — opcionalmente, armazena cópias dos estados visitados para gerar gráficos de evolução.

Trecho simplificado:

```
history.append(problem.conflicts(current))
if track_states:
    states.append(current.copy())
```

Listing 3 – Registro de histórico de conflitos e estados.

Limite de memória para visited_boards

Para manter a simplicidade e eficiência, apenas os últimos 8 estados visitados são armazenados em `visited_boards`. Estados mais antigos são descartados conforme novos são adicionados, garantindo que a verificação de ciclos curtos não consuma memória excessiva.
Em síntese, a implementação com movimentos laterais combina:

- Controle de movimentos laterais com limite configurável (*Sideways-10*, *Sideways-100*).
- Rastreamento de estados visitados para evitar ciclos curtos.
- Registro de histórico e estados para análise de desempenho e visualização.
- Flexibilidade para iniciar de um tabuleiro definido ou aleatório, garantindo comparabilidade entre execuções.

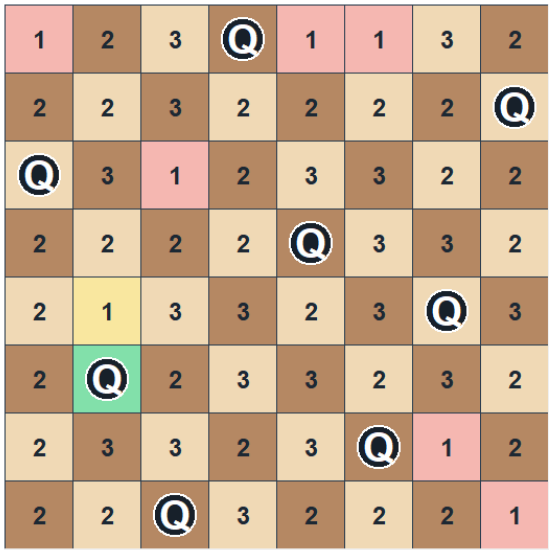
Essa estrutura permite explorar platôs de maneira eficiente, mantendo o equilíbrio entre robustez do algoritmo e simplicidade da implementação, além de facilitar a análise comparativa entre diferentes limites de movimentos laterais.

Análise Teórica

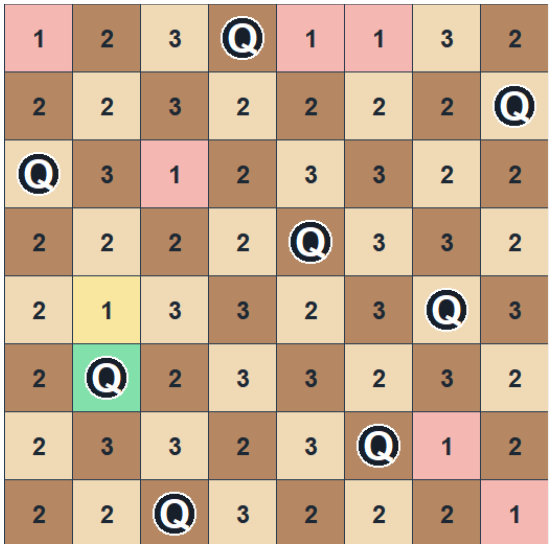
Foram analisadas duas configurações principais do método: uma com limite de 10 movimentos laterais (*Sideways-10*) e outra com limite de 100 movimentos (*Sideways-100*). Em teoria, um limite maior permite explorar mais amplamente os platôs, aumentando a probabilidade de alcançar uma solução global. No entanto, valores muito altos podem resultar em maior tempo de execução, já que o algoritmo tende a permanecer mais tempo em regiões de mesmo custo antes de progredir.

Soluções Encontradas

Ambas as variantes foram capazes de encontrar soluções corretas para o problema das 8 Rainhas, chegando a configurações finais idênticas (Figura 2). O fato de ambas convergirem para a mesma disposição reforça que a diferença entre elas está na eficiência da busca e não na qualidade das soluções encontradas.



((a)) *Sideways-10*.



((b)) *Sideways-100*.

Figura 2 – Soluções encontradas pelas variantes do algoritmo *Hill Climbing* com movimentos laterais.

Comparação de Desempenho Real

Os resultados médios das 100 execuções de cada variante são apresentados na Tabela 1. Observa-se que, embora o tempo médio de execução (*Average Time*) tenha permanecido praticamente constante entre as duas configurações (5,6 ms e 6,0 ms), a taxa de sucesso aumentou de 58% para 67% ao elevar o limite de movimentos laterais de 10 para 100. Além disso, o número médio de conflitos residuais diminuiu de 0,42 para 0,33, indicando uma melhora na qualidade das soluções alcançadas. O uso de memória foi semelhante entre os dois casos, confirmando que o aumento do limite lateral impacta mais a eficiência do processo do que os recursos computacionais.

Tabela 1 – Comparação de desempenho entre as variantes com diferentes limites de movimentos laterais.

Métrica	**Sideways-10**	**Sideways-100**
Tempo médio (ms)	5.609	5.971
Conflitos médios	0.420	0.330
Taxa de sucesso (100 exec.)	58/100	67/100
Passos médios	9.5	10.0
Pico de memória (KB)	1.266	1.215

Comparação gráfica de desempenho

A Figura 3 representa exatamente a Tabela 1 em um gráfico de barras, onde a barra azul representa o *Sideways-10* e a barra laranja o *Sideways-100*;

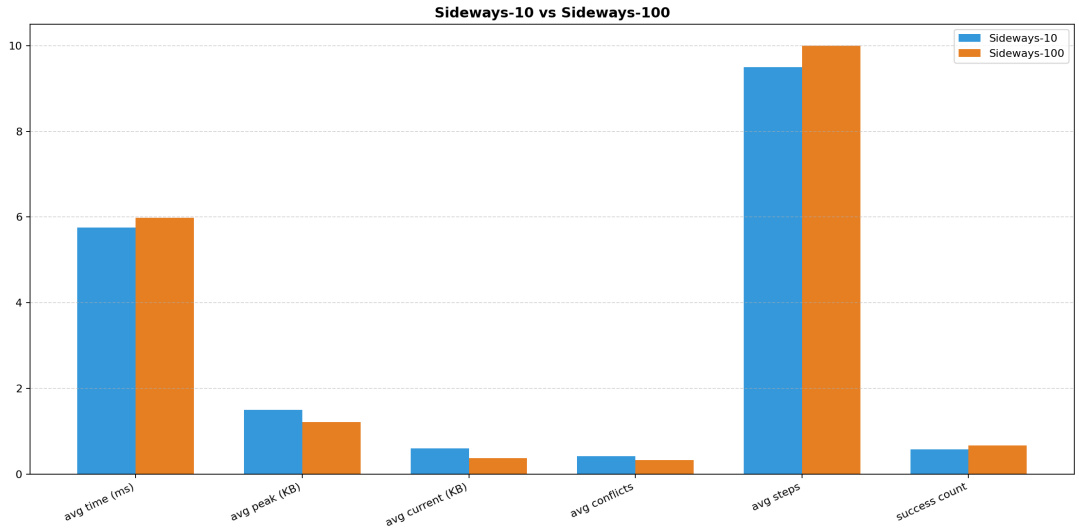
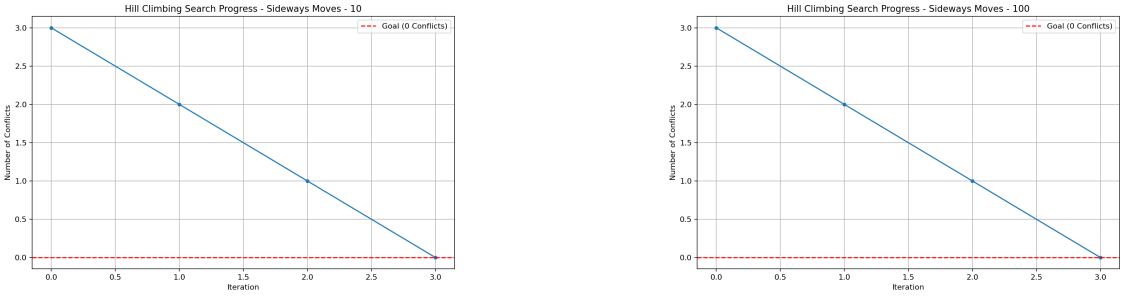


Figura 3 – Comparação gráfica de desempenho entre *Sideways-10* (azul) e *Sideways-100* (laranja).

A análise gráfica do progresso das execuções (Figura 4) mostra que ambas as variantes do algoritmo (*Sideways-10* e *Sideways-100*) apresentam reduções rápidas no número de conflitos nas primeiras iterações. No entanto, os gráficos resultaram praticamente idênticos, o que indica que o aumento no limite de movimentos laterais não alterou de forma significativa a trajetória da busca em termos de custo por iteração.

Essa semelhança ocorre porque, embora o limite maior permita que o algoritmo explore platôs por mais tempo, a configuração inicial já tende a convergir rapidamente para uma região de solução estável. Assim, as diferenças se manifestam mais nas taxas de sucesso e no número total de passos até a convergência do que na forma do gráfico.



((a)) *Sideways-10* ((b)) *Sideways-100*

Figura 4 – Evolução do número de conflitos por iteração — comparativo entre as variantes do *Hill Climbing* com movimentos laterais.

Em resumo, os gráficos confirmam que, apesar das diferenças paramétricas entre as variantes, ambas seguem trajetórias de melhoria muito próximas, reforçando a estabilidade do comportamento do algoritmo frente a pequenas variações no limite de movimentos laterais.

Hill Climbing com *Random Restarts*

O método de *Hill Climbing com Random Restarts* é uma extensão do algoritmo clássico de subida de encosta, projetado para contornar uma de suas principais limitações: a tendência de ficar preso em mínimos locais. A ideia central consiste em executar múltiplas instâncias do *Hill Climbing*, cada uma iniciando a partir de um estado inicial aleatório. Quando uma execução converge para um platô ou mínimo local, o algoritmo reinicia com uma nova configuração, aumentando significativamente a probabilidade de alcançar a solução global.
A implementação consiste em um laço externo responsável pelos reinícios, enquanto cada instância interna segue o algoritmo de *Hill Climbing* puro ou com movimentos laterais, dependendo da configuração. O número máximo de reinicializações (`max_restarts`) é definido pelo usuário, e o melhor estado encontrado entre todas as execuções é retornado como solução final.

Inicialização e configuração

O algoritmo permite receber um tabuleiro inicial ou gerar um aleatório. A função interna `_random_board()` garante que cada reinício produza uma configuração diferente. Também são inicializados:

- **Melhor solução até o momento** (`best_solution`) — armazena o estado com menor número de conflitos.
- **Melhor fitness** (`best_fitness`) — controla a qualidade da melhor solução.
- **Contador de reinícios** (`restart_count`) — acompanha quantas reinicializações foram realizadas.
- **Histórico e estados** (`best_history`, `best_states`) — opcionais para análise gráfica e acompanhamento da evolução da busca.

Estrutura do laço de reinícios

O algoritmo executa um laço `for` para cada reinício, gerando um novo tabuleiro inicial ou utilizando o fornecido na primeira execução. A cada iteração:

- Se `allow_sideways` estiver habilitado, é utilizado o *Hill Climbing com Sideways Moves*.
- Caso contrário, utiliza-se o *Hill Climbing* puro.
- Após a execução, o fitness do estado final é comparado com o melhor já encontrado. Se superior, os registros são atualizados.

- A busca pode ser interrompida prematuramente se uma solução ótima (zero conflitos) for encontrada.

Trecho central do laço de reinícios:

```
best_solution = None
best_fitness = float('-inf')
for restart in range(max_restarts):
if restart == 0 and initial_board is not None:
start_board = list(initial_board)
else:
start_board = _random_board()  # gera tabuleiro aleatorio

```
if allow_sideways:
    current, history, states = hill_climbing_with_sideways_moves(
        problem,
        max_moves_per_restart,
        track_states=track_states,
        initial_board=start_board,
)
else:
    current, history, states = hill_climbing(
        problem,
        track_states=track_states,
        initial_board=start_board,
)

current_fitness = problem.fitness(current)
if current_fitness > best_fitness:
    best_fitness = current_fitness
    best_solution = current
    best_history = history
    best_states = states

if best_fitness == 0: # encontrou solucao otima
    break
```
```

Listing 4 – Laço de reinícios e escolha do melhor estado.

Geração de tabuleiros aleatórios

Cada reinício utiliza uma função para gerar uma configuração inicial randômica. Isso garante diversidade no espaço de busca:

```
def _random_board() -> List[int]:
board = list(range(8))
```

```
generator.shuffle(board)
return board
```

Listing 5 – Geração de tabuleiros aleatórios para reinícios.

A aleatoriedade permite que o algoritmo explore diferentes regiões do espaço de soluções, evitando armadilhas de mínimos locais.

Rastreamento e histórico

Assim como no *Sideways Moves*, o histórico de conflitos e os estados visitados podem ser armazenados opcionalmente. Isso permite a análise detalhada da evolução de cada reinício e da melhor solução encontrada.

```
best_history = history
best_states = states if track_states else None
```

Listing 6 – Registro do histórico de conflitos e estados da melhor execução.

O *Hill Climbing com Random Restarts* combina:

- Múltiplas execuções a partir de tabuleiros iniciais distintos, aumentando a probabilidade de encontrar a solução global.
- Flexibilidade para utilizar *Hill Climbing* puro ou com movimentos laterais em cada reinício.
- Registro de histórico e estados para análise comparativa e geração de gráficos.
- Interrupção prematura caso a solução ótima seja encontrada, otimizando o tempo de execução.

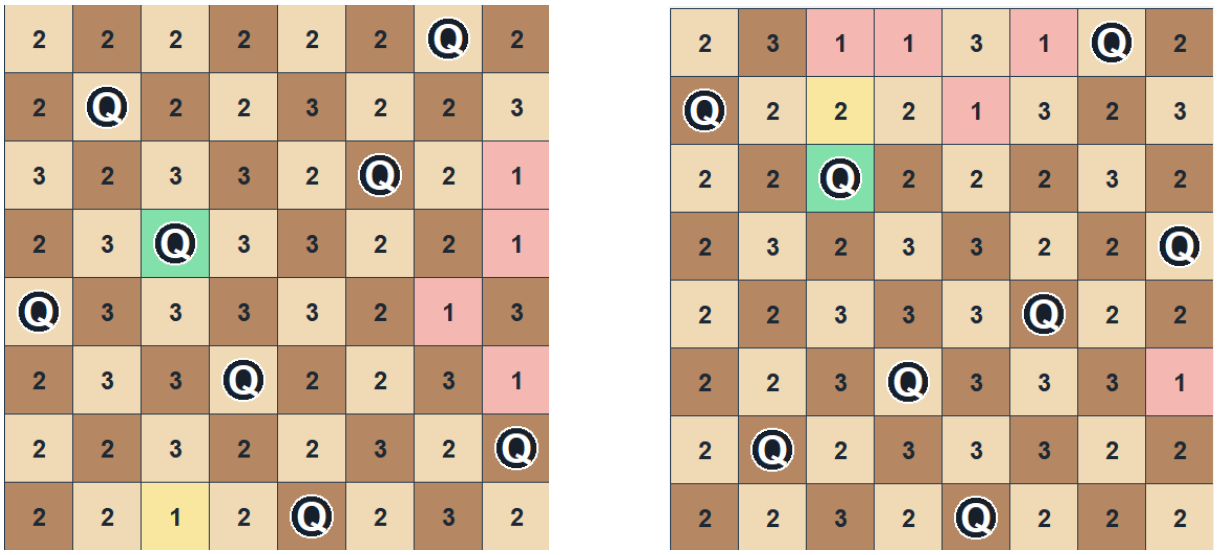
Essa implementação permite explorar o espaço de busca de forma robusta, tornando o algoritmo resistente à qualidade inicial do tabuleiro e mantendo o equilíbrio entre eficiência e simplicidade na codificação.

Análise Teórica

A introdução dos reinícios aleatórios transforma o comportamento do algoritmo: em vez de uma busca determinística em torno de um único ponto, ele realiza múltiplas amostragens do espaço de estados. Dessa forma, a probabilidade de encontrar o estado ótimo aumenta exponencialmente com o número de reinicializações, desde que o espaço de busca não seja excessivamente grande e que o método base (*Hill Climbing* puro ou com movimentos laterais) seja eficiente em explorar vizinhanças locais.

Soluções Encontradas

As soluções finais obtidas pelas duas variantes — uma utilizando o *Hill Climbing* simples e outra com movimentos laterais — diferem visualmente, ainda que ambas atinjam um estado ótimo (sem conflitos). Essa diferença decorre da natureza aleatória das reinicializações e do comportamento distinto de cada estratégia durante a busca (Figuras 5(a) e 5(b)).



((a)) *Random Restarts Hill Climbing* ((b)) *Random Restarts Sideways Moves*

Figura 5 – Soluções finais obtidas pelas variantes com reinicializações aleatórias.

Comparação de Desempenho Real

A Tabela 2 e a Figura 6 apresentam o comparativo entre as variantes com reinicialização. Ambas alcançaram 100% de sucesso, evidenciando a eficácia do uso de múltiplos pontos de partida. No entanto, a versão baseada no *Hill Climbing* puro apresentou maior tempo médio de execução, enquanto a versão com movimentos laterais se mostrou mais eficiente e estável, alcançando resultados equivalentes com menor custo temporal.

Tabela 2 – Comparação de desempenho entre as variantes de *Hill Climbing* com reinicializações aleatórias.

Métrica	**RandomRestartsHill**	**RandomRestartsSideways**
Tempo médio (ms)	21.084	8.719
Conflitos médios	0.000	0.000
Taxa de sucesso	100/100	100/100
Passos médios	4.4	9.6
Pico de memória (KB)	1.746	1.793

Comparação gráfica de desempenho

A Figura 6 representa graficamente os dados da Tabela 2. A barra azul indica a variante *Random Restarts Hill*, enquanto a barra laranja representa o *Random Restarts Sideways*.

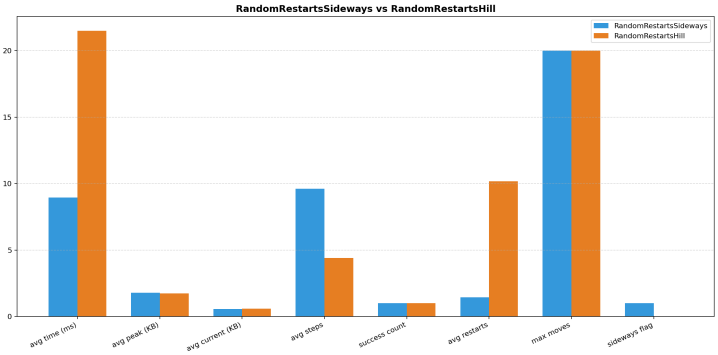
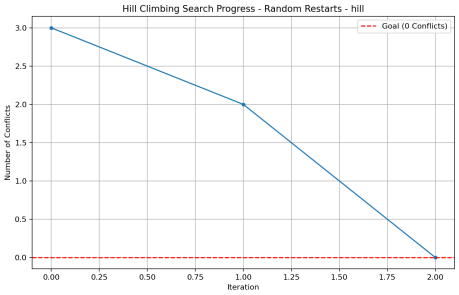
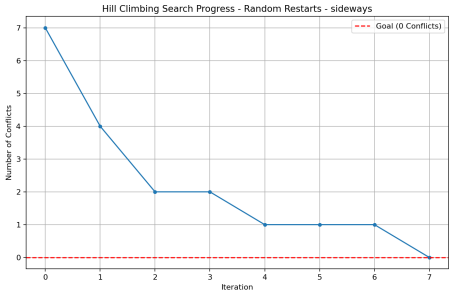


Figura 6 – Comparação gráfica de desempenho entre as variantes de *Random Restarts*.

A análise gráfica do progresso das execuções (Figura 7) revela comportamentos distintos entre as duas abordagens. No caso do *Random Restarts Hill*, a solução é alcançada em apenas três passos — iniciando com três conflitos, reduzindo para dois e chegando a zero na terceira iteração. Já o *Random Restarts Sideways* percorre um caminho mais gradual, necessitando de sete iterações, com reduções sucessivas de $7 \rightarrow 4 \rightarrow 2 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 0$ conflitos.
Essa diferença ocorre porque, com os movimentos laterais permitidos, o algoritmo tende a explorar mais amplamente regiões de platô antes de aceitar um reinício, o que aumenta o número médio de passos, mas reduz o tempo total, já que evita reinicializações prematuras.



((a)) *Random Restarts Hill Climbing*



((b)) *Random Restarts Sideways Moves*

Figura 7 – Evolução do número de conflitos por iteração — comparativo entre as variantes com *Random Restarts*.

Em síntese, o uso de reinicializações aleatórias garante convergência total em todos os testes, eliminando a dependência de boas condições iniciais. A combinação com movimentos laterais mantém um equilíbrio ideal entre desempenho e eficiência computacional, explorando melhor os platôs e obtendo uma trajetória de busca mais estável e consistente.

Simulated Annealing

O algoritmo *Simulated Annealing* é uma extensão probabilística do método de *Hill Climbing*, inspirado no processo físico de recozimento de metais (*annealing*). Diferentemente do *Hill Climbing* puro, que aceita apenas movimentos que melhoram o estado, o *Simulated Annealing* permite, sob certas condições, a aceitação de soluções piores, com o objetivo de escapar de mínimos locais e aumentar a probabilidade de encontrar uma solução globalmente ótima.
A aceitação de piores soluções é governada por uma temperatura T que decresce ao longo das iterações de acordo com uma função de resfriamento (*cooling schedule*). Inicialmente, $T_0 = 400$, garantindo liberdade de exploração, e a temperatura é reduzida gradualmente ao longo de 1000 iterações, podendo seguir um decaimento linear ou exponencial.

Representação do estado e avaliação

O estado do problema continua sendo representado por um vetor de 8 posições, indicando a linha ocupada por cada rainha. O operador de vizinhança realiza deslocamentos verticais individuais de cada rainha, e a função de avaliação computa o número de pares de rainhas em conflito.

Aceitação probabilística de piores estados

Para cada vizinho selecionado aleatoriamente, o algoritmo compara o número de conflitos:

- **Melhor que o atual**: o vizinho é adotado imediatamente.
- **Pior que o atual**: o vizinho pode ser aceito com probabilidade

$$P = e^{-\Delta E/T},$$

onde ΔE é a diferença entre o número de conflitos do estado atual e do candidato. Um número aleatório $r \in [0,1]$ é gerado, e o movimento é aceito se $r < P$.

Trecho ilustrativo do núcleo do algoritmo:

```
delta_conflicts = current_conflicts - candidate_conflicts
if delta_conflicts > 0:
    current = candidate  # melhora
else:
    acceptance = math.exp(delta_conflicts / T)
    r = random.uniform(0, 1)
    if r < acceptance:
        current = candidate  # aceita piora
```

Listing 7 – Aceitação de vizinhos no Simulated Annealing.

Função de resfriamento (cooling schedule)

O decaimento da temperatura é implementado pela função `schedule`, que permite alternar entre:

- **Linear**: $T = 1 - \frac{t+1}{\text{max\_steps}}$
- **Exponencial**: $T = T \cdot 0.99$

Trecho do código:

```
def schedule(t, cooling_func, initial_temp=400, max_steps=1000):
    if cooling_func == 1:
        val = 1 - ((t+1) / max_steps)  # Linear
        return val if val > 0 else 0
    elif cooling_func == 2:
        val = initial_temp * 0.99       # Exponencial
        return val if val > 0 else 0
```

Listing 8 – Função de resfriamento da temperatura.

Rastreamento de histórico e estados

Para análise posterior e geração de gráficos de evolução, o algoritmo mantém:

- `history` — lista do número de conflitos a cada iteração.
- `states` — opcionalmente, cópias dos estados visitados.

Trecho simplificado:

```
history.append(problem.conflicts(current))
if track_states and states is not None:
    states.append(current.copy())
```

Listing 9 – Registro de histórico e estados.

O *Simulated Annealing* combina:

- Aceitação probabilística de piores soluções para escapar de mínimos locais.
- Controle da exploração por meio da temperatura, reduzida segundo funções de resfriamento linear ou exponencial.
- Representação de estados e operadores de vizinhança compatíveis com as demais variantes do *Hill Climbing*.
- Registro de histórico e estados para análise detalhada da evolução da busca.

A flexibilidade da temperatura e a aceitação controlada de movimentos piores tornam o *Simulated Annealing* robusto frente a armadilhas de mínimos locais, equilibrando exploração e convergência.

Análise Teórica

Foram avaliadas duas variações principais do *Simulated Annealing*: uma com resfriamento linear e outra com resfriamento exponencial. Em teoria, a versão linear proporciona uma redução uniforme da temperatura, favorecendo uma exploração mais previsível e controlada. Já a exponencial reduz a temperatura de forma mais agressiva, permitindo uma rápida convergência, mas também maior risco de aprisionamento prematuro em mínimos locais. Ambas compartilham o mesmo limite máximo de iterações (1000), o que possibilita uma comparação direta entre suas eficiências e dinâmicas de busca.

Soluções Encontradas

As duas variantes convergiram para tabuleiros finais sem conflitos, mostrando que o método é capaz de encontrar soluções completas mesmo sob funções de resfriamento distintas. As Figuras 8 e 9 mostram os tabuleiros resolvidos para cada caso.

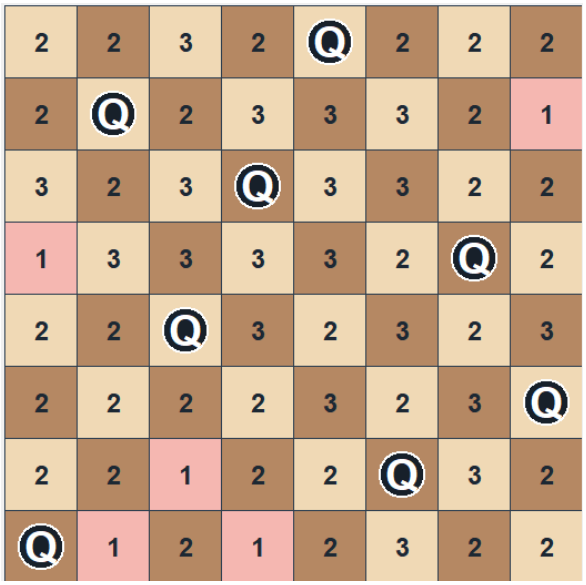


Figura 8 – Tabuleiro resolvido — *Simulated Annealing Linear*.

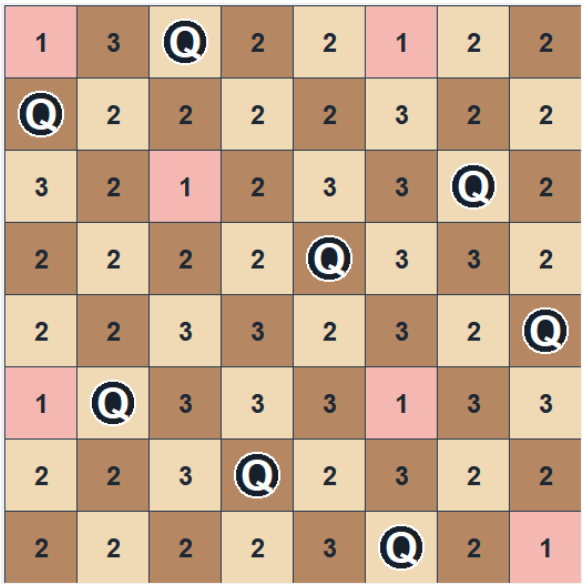


Figura 9 – Tabuleiro resolvido — *Simulated Annealing Exponential*.

Comparação de Desempenho Real

A Tabela 3 resume o desempenho médio das duas versões. Ambas atingiram taxas de sucesso próximas (44% e 45%), indicando estabilidade e robustez semelhantes. O tempo médio de execução e o número de passos até a convergência também foram praticamente idênticos, refletindo a convergência equivalente das funções de resfriamento para o mesmo número de iterações.

Tabela 3 – Comparação de desempenho entre variantes do *Simulated Annealing*.

Métrica	**Exponential**	**Linear**
Tempo médio (ms)	61.809	61.951
Conflitos médios	0.560	0.550
Taxa de sucesso	44/100	45/100
Passos médios	1001.0	1000.0
Pico de memória (KB)	19.123	19.196

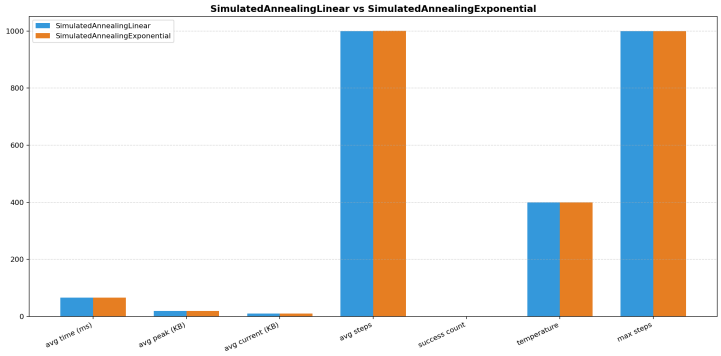


Figura 10 – Comparação gráfica de desempenho entre funções de resfriamento.

Comparação Gráfica de Desempenho

As Figuras 11 e 12 mostram a evolução do número de conflitos por iteração para cada função de resfriamento. Ambas apresentam padrões quase idênticos, com reduções rápidas nos primeiros ciclos e estabilização conforme a temperatura diminui. Essa semelhança se deve ao fato de que, com o número máximo de passos fixado em 1000 e temperaturas iniciais elevadas, as curvas de decaimento acabam convergindo para magnitudes próximas de temperatura ao longo da execução. Dessa forma, o comportamento dinâmico do algoritmo é mantido, e o tipo de resfriamento — linear ou exponencial — não influencia significativamente o resultado final, apenas o ritmo de exploração nas primeiras iterações.

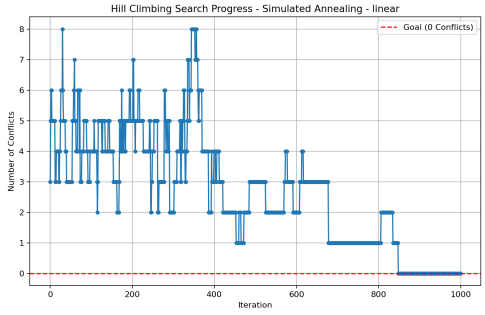


Figura 11 – Evolução dos conflitos — *Simulated Annealing Linear*.

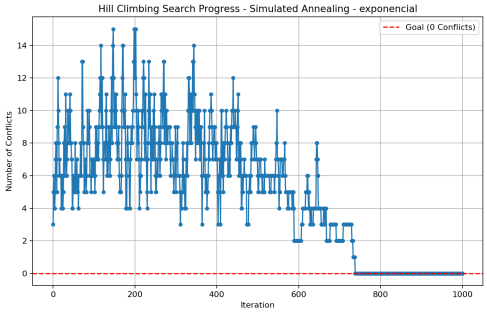


Figura 12 – Evolução dos conflitos — *Simulated Annealing Exponential*.

Em resumo, os resultados mostram que o *Simulated Annealing* apresentou estabilidade e boa capacidade de escapar de mínimos locais, embora com desempenho inferior às variantes com reinício aleatório, principalmente devido à natureza estocástica e ao custo adicional de simular o resfriamento gradual.

Comparação Geral dos Algoritmos

Após a análise individual de cada algoritmo, é importante comparar seu desempenho de forma global, considerando tempo de execução, número médio de conflitos, taxa de sucesso, número médio de passos e uso de memória.

Para esta análise, o programa foi executado novamente para obter os dados comparativos, razão pela qual alguns valores diferem ligeiramente das seções anteriores. No entanto, como se trata de médias de 100 execuções, as diferenças são mínimas e não afetam as conclusões.

A Tabela 4 apresenta os resultados globais de todas as variantes analisadas, incluindo o importante consumo médio de memória corrente (*Average Current (KB)*), que indica o uso real de memória durante a execução.

Métrica	RandomRestartsHill	RandomRestartsSideways	Sideways-10	Sideways-100	SA Exp.	SA Lin.
Average Time (ms)	21.279	8.955	5.742	6.141	65.321	65.398
Average Current (KB)	0.590	0.567	0.441	0.374	10.046	10.315
Average Conflicts	0.000	0.000	0.420	0.330	0.560	0.550
Success Rate	100/100	100/100	58/100	67/100	44/100	45/100
Average Steps	4.4	9.6	9.5	10.0	1001.0	1000.0
Peak Memory (KB)	1.746	1.793	1.266	1.215	19.123	19.196

Tabela 4 – Resultados gerais atualizados dos algoritmos, incluindo memória corrente média.

Análise de Tempo de Execução

Os algoritmos de *Hill Climbing* puros (com ou sem movimentos laterais) apresentam tempos médios de execução significativamente menores, variando entre 5,742 ms e 21,279 ms. O *Simulated Annealing*, por outro lado, apresenta tempos superiores a 65 ms, devido à avaliação probabilística e à necessidade de iterar por 10.000 passos, independentemente de melhorias imediatas.

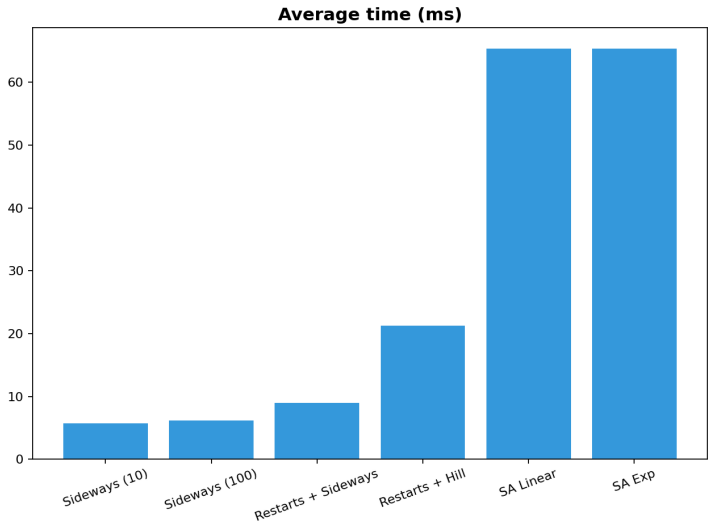


Figura 13 – Comparação do tempo médio de execução entre algoritmos.

Número Médio de Conflitos

Os algoritmos com reinícios aleatórios (*Random Restarts*) alcançam consistentemente 0 conflitos, demonstrando robustez na resolução completa do problema. Já os algoritmos *Sideways* e *Simulated Annealing* apresentam uma média de conflitos próxima de 0,33–0,56, refletindo a presença de platôs ou mínimos locais não totalmente resolvidos.

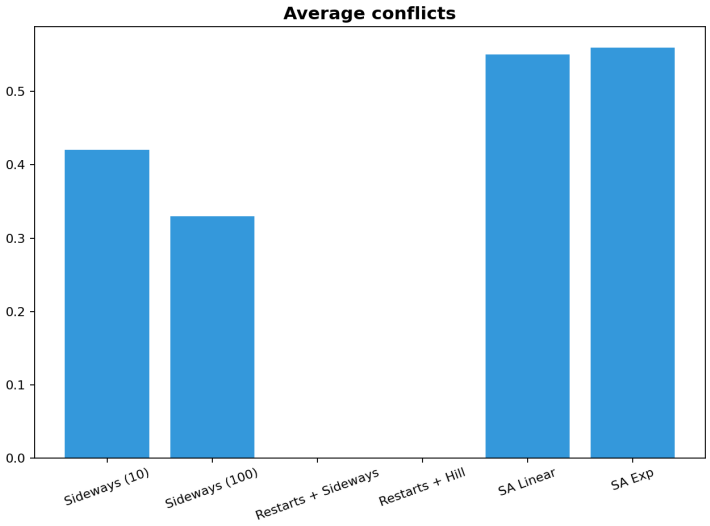


Figura 14 – Média de conflitos ao final da execução.

Taxa de Sucesso

A taxa de sucesso (número de execuções que atingiram solução ótima) confirma o padrão observado nos conflitos: os métodos com reinício aleatório atingem 100% de sucesso, enquanto *Sideways* e *Simulated Annealing* variam entre 44% e 67%, mostrando maior sensibilidade a mínimos locais.

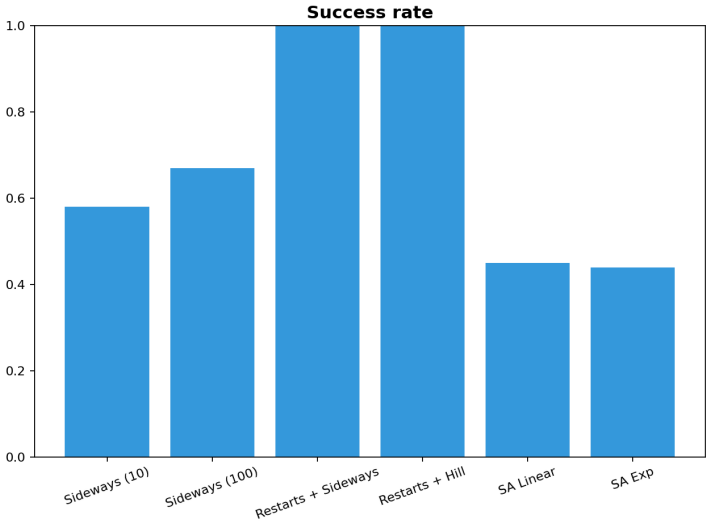


Figura 15 – Taxa de sucesso de cada algoritmo.

Número Médio de Passos e Memória

O algoritmo *Simulated Annealing* realiza muito mais passos devido à exploração adicional: o *Simulated Annealing* atinge cerca de 1000 passos por execução, enquanto os métodos de *Hill Climbing* variam entre 4,4 e 10 passos.

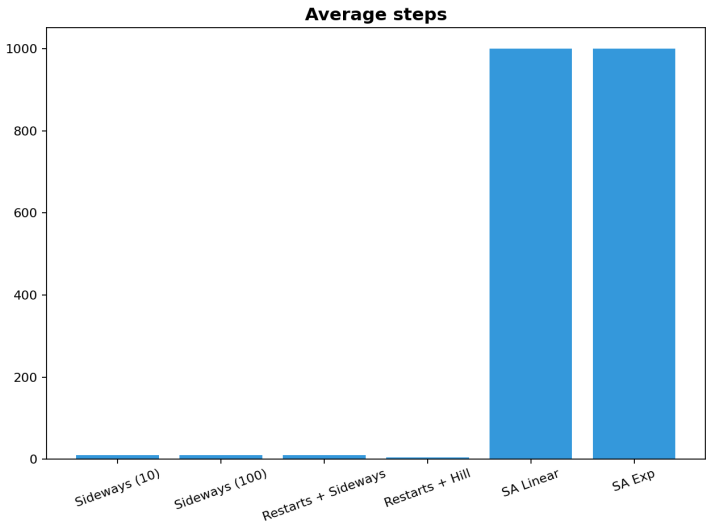


Figura 16 – Número médio de passos por execução.

Memória Corrente Média

O gráfico da Figura 17 mostra o consumo médio de memória em tempo real (*Average Current*) durante a execução. Os algoritmos de *Hill Climbing* puros e com movimentos laterais utilizam muito pouca memória (0,374–0,590 KB), refletindo a simplicidade de armazenar apenas o estado atual e um histórico limitado. O *Simulated Annealing* consome cerca de 10 KB, devido ao armazenamento contínuo dos estados visitados para gerar gráficos de evolução e registrar a trajetória da busca.

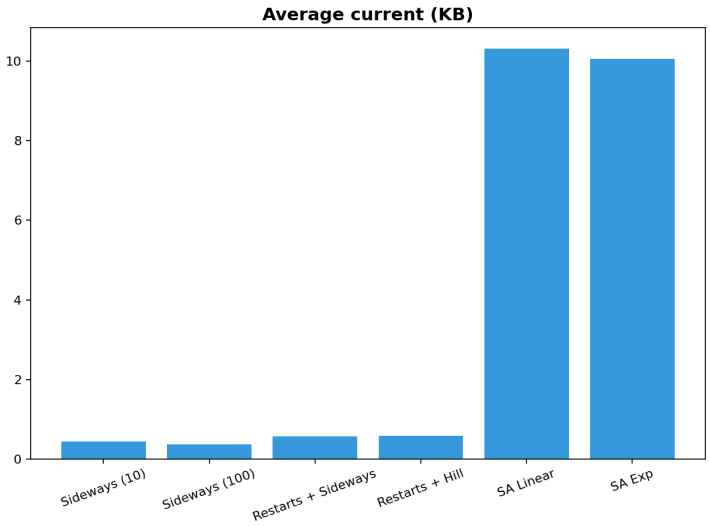


Figura 17 – Memória corrente média utilizada pelos algoritmos.

Memória Máxima

A memória máxima (*Peak Memory*) segue padrão semelhante: *Simulated Annealing* atinge valores próximos de 19 KB, enquanto *Hill Climbing* permanece abaixo de 2 KB, refletindo a eficiência em armazenamento durante a execução.

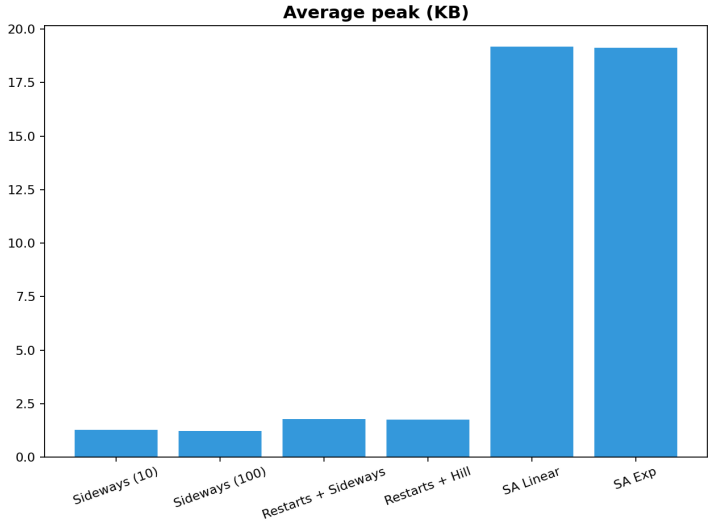


Figura 18 – Memória máxima utilizada pelos algoritmos.

A comparação geral evidencia claramente os trade-offs entre os métodos:

- *Random Restarts* é mais eficiente em termos de tempo, garante 100% de sucesso e utiliza pouca memória.
- Movimentos laterais (*Sideways*) melhoram a taxa de sucesso em relação ao *Hill Climbing* puro, mas ainda apresentam falhas ocasionais e mantêm consumo de memória muito baixo.
- *Simulated Annealing* explora mais amplamente o espaço de busca, mas com custo maior de tempo e memória, sem garantir solução ótima em todas as execuções.

Conclusão

A análise comparativa dos algoritmos estudados evidencia diferenças claras entre as estratégias de busca utilizadas para resolver o problema das 8 rainhas. Os métodos de *Hill Climbing* com reinícios aleatórios se mostraram os mais robustos, garantindo 100% de sucesso em todas as execuções e mantendo o número médio de conflitos igual a zero. Essa abordagem se beneficia da combinação entre simplicidade do *Hill Climbing* e a variabilidade proporcionada pelos reinícios, permitindo escapar de mínimos locais de forma eficaz, com tempo de execução relativamente baixo e consumo de memória reduzido.

A introdução de movimentos laterais (*Sideways Moves*) apresentou melhorias em relação ao *Hill Climbing* puro, especialmente na capacidade de explorar platôs, mas não garantiu sucesso completo. As variantes *Sideways-10* e *Sideways-100* alcançaram taxas de sucesso intermediárias, variando entre 58% e 67%, com média de conflitos próxima de 0,33–0,42. Embora o aumento do limite de movimentos laterais melhore a taxa de sucesso, também resulta em ligeiro aumento do tempo médio de execução, mostrando o trade-off entre exploração e eficiência temporal.

O *Simulated Annealing*, por sua vez, demonstrou ser capaz de explorar o espaço de busca de maneira mais ampla, permitindo transições para estados de menor qualidade e oferecendo maior flexibilidade na superação de mínimos locais. No entanto, essa exploração mais agressiva implicou em custos computacionais mais altos, com tempos médios de execução superiores a 65 ms, consumo de memória corrente próximo de 10 KB e memória máxima ao redor de 19 KB. A taxa de sucesso, de 44% a 45%, e o número médio de conflitos (aproximadamente 0,55–0,56) indicam que, embora o algoritmo explore efetivamente, ele não garante soluções ótimas em todas as execuções.

Em síntese, a escolha do algoritmo depende do equilíbrio desejado entre robustez, eficiência e consumo de recursos. Os métodos com reinícios aleatórios são ideais quando se busca garantia de solução com baixo custo computacional, enquanto os movimentos laterais podem ser úteis em cenários onde a exploração de platôs é relevante. Já o *Simulated Annealing* se mostra mais apropriado para problemas que exigem exploração ampla do espaço de busca, sendo eficaz em evitar mínimos locais, mas com maior demanda de tempo e memória. Dessa forma, o estudo fornece uma visão clara das vantagens e limitações de cada abordagem, permitindo selecionar a estratégia mais adequada conforme o contexto do problema.

Créditos e Declaração de Autoria

Autores e Papéis

Guilherme Alvarenga de Azevedo:

1. **Implementação dos Algoritmos**:
    - **Busca Local**: *Sideways Moves*, *Random Restart*, *Simulated Annealing*;
    - **Experimentos**: Função principal + Interação com Terminal. Medição de tempo/memória e Início da Comparação dos Algoritmos;
    - **Visualização**: GUI do programa, geração de alguns dos GIFs e grafos (corrigido/melhorado com IA);
2. **README.md**: Redigiu o `README.md` do projeto que se encontra no diretório referente a esse trabalho (`/ia-trabalhos/trabalho2`). Além de colocar as dependências do projeto no `requirements.txt`.
3. **Relatório**:
    - Realizou parte do capítulo de *Problema*, descrevendo a boa parte da seção *Organização e Execução do Projeto*;
    - Corrigiu detalhes no relatório como um todo;

Maria Eduarda Teixeira Souza:

1. **Implementação dos Algoritmos**:
    - **Experimentos**: Algumas correções da função principal + interação com Terminal. Melhorias nos algoritmos de comparação;
    - **Visualização**: Melhoria das representações em GIFs, geração da maior parte dos gráficos de métricas e de comparação (corrigido/melhorado com IA).
2. **Relatório**:
    - No capítulo *Problema* desenvolveu as seções de *Descrição*, *Modelagem*, *Algoritmos* e *Critério de Análise e Avaliação*;
    - Redigiu os capítulos de cada algoritmo, detalhando informações sobre seus funcionamentos e suas implementações;
    - Realizou as comparações e redigiu o capítulo dessa parte;
    - Redigiu a *Conclusão* do trabalho.

Uso de IA

1. **Adição de comentários**: *Copilot*;
2. **Visualização em GIFs**: Melhorias/Correções - *ChatGPT*, *Copilot*;
3. **GUI do programa**: Melhorias/Correções - *ChatGPT*, *Copilot*;

Referências - Recursos Externos

[1] AZEVEDO, Guilherme A. SOUZA, Maria E. T. **IA-Trabalhos**: Trabalho 2 - Busca Local. 2025. Disponível em: <https://github.com/dudatsouza/ia-trabalhos> Acesso em: 04 nov. 2025.
[2] OLIVEIRA, Tiago A. **Inteligência Artificial 04**: Estruturas e Estratégias de Busca - Busca em Ambientes Complexos. Slides de Aula. 2025.
[3] RUSSELL, Stuart J; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna**. 4. ed. Pearson, 2021.

Declaração

Confirmamos que o código entregue foi desenvolvido pela equipe, respeitando as políticas da disciplina. As ferramentas de Inteligência Artificial generativa foram utilizadas para comentar o código já feito e, também, melhorar partes que não seriam avaliadas de acordo com o enunciado do trabalho. Tais como a **Visualização em GIFs** e **GUI do programa**.