

# Trabalho 01 — Algoritmos de Busca

[201804940002] Eduardo Gil S. Cardoso      [201804940016] Gabriela S. Maximino  
[201704940007] Igor Matheus S. Moreira

4 de dezembro de 2020

Este trabalho é referente à disciplina de Inteligência Artificial do curso de Bacharelado em Ciência da Computação na Universidade Federal do Pará. Ele propõe a implementação de uma versão do algoritmo *Stochastic Hill Climbing (Simulated Annealing)* e de uma meta-heurística (*Genetic Algorithm*) para a resolução do **problema das oito rainhas**.

**Observação:** antes de interagir com o código no Jupyter Notebook/Lab, é importante observar que certas células anteriores à que se quer executar podem ser necessárias. A fim de evitar isso, é importante certificar-se de executar ao menos uma vez todas as células contendo definições de funções ou importações de módulos.

## Sumário

1. Requisitos
  1. Ambiente
  2. Funções de checagem
2. Modelagem do tabuleiro, da função-objetivo e das soluções
  1. Funções auxiliares relacionadas ao tabuleiro
  2. A classe Tabuleiro
  3. A função-objetivo
  4. As soluções
3. Outras definições globais
  1. Heurísticas adotadas
  2. Critérios de parada adotados
4. Problemas
  1. Stochastic Hill Climbing: Simulated Annealing
    1. Implementação do SA
      1. Funções auxiliares ao SA
      2. Algoritmo principal SA
    2. Heurísticas e critérios de parada do SA
    3. Resultados do SA
  2. Meta-heuristic: Genetic Algorithm
    1. Implementação do GA
      1. Funções auxiliares ao GA
      2. Algoritmo principal GA
    2. Operadores do GA
    3. Resultados do GA
5. Considerações finais

# 1 Requisitos

## 1.1 Ambiente

Este trabalho foi feito utilizando a seguinte linguagem de programação:

- python 3.8.5

Em adição, os seguintes módulos precisam estar instalados no ambiente em que este notebook for executado:

- ipypublish 0.10.12
- matplotlib 3.3.2
- numpy 1.19.2
- pandas 1.1.4
- scipy 1.5.3
- seaborn 0.11.0

Observe que o código pode funcionar em versões distintas às que foram mencionadas acima; contudo, sabe-se que a sua execução é garantida nas versões mencionadas.

```
[1]: import decimal as d
import numpy as np
import random as r
import seaborn as sns
import typing as t

from ipypublish import nb_setup
from scipy import stats
from timeit import default_timer as timer
```

```
[2]: %%capture

plt = nb_setup.setup_matplotlib(output=('pdf',), usetex=False, rcparams={'axes.
→facecolor': 'white', 'figure.facecolor': 'white'})
pd = nb_setup.setup_pandas(escape_latex=True)
```

## 1.2 Funções de checagem

Uma vez configurado o ambiente em que este notebook será executado, é necessário definir algumas funções de checagem que serão utilizadas mais adiante:

```
[3]: def verifica_comprimento_binario_igual_a(**parametros):
    numero_de_parametros = len(parametros.keys())

    if numero_de_parametros != 2:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro, outro_parametro = parametros.keys()
```

```

valor, descricao = parametros[parametro]
outro_valor, outra_descricao = parametros[outro_parametro]

if outro_valor is not None and len(valor) != outro_valor:
    raise ValueError(f"O comprimento do {descricao} {parametro}, em binário,
→ precisa ser igual ao {outra_descricao} {outro_parametro}.")

```

```

[4]: def verifica_comprimento_maior_ou_igual_a(**parametros):
    numero_de_parametros = len(parametros.keys())

    if numero_de_parametros != 2:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro, outro_parametro = parametros.keys()

    valor, descricao = parametros[parametro]
    outro_valor, outra_descricao = parametros[outro_parametro]

    if outro_valor is not None and len(valor) < outro_valor:
        raise ValueError(f"O {descricao} {parametro} precisa receber um valor,
→ no mínimo, igual ao {outra_descricao} {outro_parametro}.")

```

```

[5]: def verifica_comprimento_menor_ou_igual_a(**parametros):
    numero_de_parametros = len(parametros.keys())

    if numero_de_parametros != 2:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro, outro_parametro = parametros.keys()

    valor, descricao = parametros[parametro]
    outro_valor, outra_descricao = parametros[outro_parametro]

    if outro_valor is not None and len(valor) > outro_valor:
        raise ValueError(f"O comprimento do {descricao} {parametro} precisa ser,
→ no máximo, igual ao {outra_descricao} {outro_parametro}.")

```

```

[6]: def verifica_dtype(**parametro_dict):
    numero_de_parametros = len(parametro_dict.keys())

    if numero_de_parametros != 1:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

```

```

parametro = list(parametro_dict.keys())[0]
valor, descricao, dtype = parametro_dict[parametro]

if dtype == np.int_ and valor.dtype != dtype:
    if valor.dtype == np.float_:
        return valor.astype(np.int_)
    else:
        raise TypeError(f"O {descricao} {parametro} precisa ser um numpy_
→array com atributo dtype igual a {dtype}. O dtype do numpy array recebido é_
→{valor.dtype}.")

if valor.dtype != dtype:
    raise TypeError(f"O {descricao} {parametro} precisa ser um numpy array_
→com atributo dtype igual a {dtype}. O dtype do numpy array recebido é {valor.
→dtype}.")
else:
    return valor

```

```

[7]: def verifica_maior_ou_igual_a(**parametros):
    numero_de_parametros = len(parametros.keys())

    if numero_de_parametros != 2:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro, outro_parametro = parametros.keys()

    valor, descricao = parametros[parametro]
    outro_valor, outra_descricao = parametros[outro_parametro]

    if outro_valor is not None and valor < outro_valor:
        raise ValueError(f"O {descricao} {parametro} precisa receber um valor,
→no mínimo, igual ao {outra_descricao} {outro_parametro}.")

```

```

[8]: def verifica_menor_ou_igual_a(**parametros):
    numero_de_parametros = len(parametros.keys())

    if numero_de_parametros != 2:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro, outro_parametro = parametros.keys()

    valor, descricao = parametros[parametro]
    outro_valor, outra_descricao = parametros[outro_parametro]

    if outro_valor is not None and valor > outro_valor:

```

```
        raise ValueError(f"O {descricao} {parametro} precisa receber um valor,
→no máximo, igual ao {outra_descricao} {outro_parametro}.")
```

```
[9]: def verifica_nao_negatividade(**parametros):
    for parametro in parametros.keys():
        valor, descricao = parametros[parametro]

        if valor < 0:
            raise ValueError(f"O {descricao} {parametro} precisa receber um
→número não-negativo.")
```

```
[10]: def verifica_ndim(**parametros):
    for parametro in parametros.keys():
        valor, descricao, ndim = parametros[parametro]

        if valor.ndim != ndim:
            raise ValueError(f"O o atributo ndim do {descricao} {parametro}
→precisa ser igual a {ndim}.")
```

```
[11]: def verifica_tipo(**parametro_dict):
    numero_de_parametros = len(parametro_dict.keys())

    if numero_de_parametros != 1:
        raise ValueError(f"Apenas um parâmetro pode ser passado para esta função.
→ Foram recebidos {numero_de_parametros}.")

    parametro = list(parametro_dict.keys())[0]
    valor, descricao, tipos = parametro_dict[parametro]

    if tipos == t.SupportsFloat:
        if not isinstance(valor, tipos):
            raise TypeError(f"O {descricao} {parametro} precisa receber um
→número de ponto flutuante ou um objeto que possa ser convertido para tal.")
        else:
            return float(valor)

    if tipos == t.SupportsInt:
        if not isinstance(valor, tipos):
            raise TypeError(f"O {descricao} {parametro} precisa receber um
→número inteiro ou um objeto que possa ser convertido para tal.")
        else:
            return int(valor)

    if tipos == np.ndarray:
        if not isinstance(valor, np.ndarray):
            if not isinstance(valor, (list, tuple)):
```

```

        raise TypeError(f"O {descricao} {parametro} precisa receber um_
→array numpy ou um objeto que possa ser convertido para tal.")
    else:
        return np.array(valor)

    if tipos == bool:
        if not isinstance(valor, bool):
            if isinstance(valor, np.bool_):
                return bool(valor)
            else:
                raise TypeError(f"O {descricao} {parametro} precisa receber um_
→objeto booleano ou um objeto que possa ser convertido para tal.")
        else:
            return valor

    if not isinstance(valor, tipos):
        raise TypeError(f"O {descricao} {parametro} precisa receber um objeto de_
→classe {tipos} ou que herde dela.")
    else:
        return valor

```

```

[12]: def verifica_tipo_operador(operador, valor, tipo):
        if not isinstance(valor, tipo):
            raise TypeError(f"O operador '{operador}' precisa ser do tipo {tipo}.")

```

## 2 Modelagem do tabuleiro, da função-objetivo e das soluções

Uma vez que ambos os algoritmos a serem discutidos aqui – *Simulated Annealing* e *Genetic Algorithm* — são utilizados para resolver o mesmo problema – o problema das oito rainhas –, optou-se por definir uma classe comumente utilizada por ambos os algoritmos, intitulada Tabuleiro.

### 2.1 Funções auxiliares relacionadas ao tabuleiro

Antes de definirmos a classe Tabuleiro, precisamos definir algumas funções para auxiliar na representação e nas operações da classe quando a codificação binária estiver ativada.

```

[13]: def ajusta_indentacao(string, string_para_adicionar):
        string = verifica_tipo(string=(string, "parâmetro", str))
        string_para_adicionar =_
→verifica_tipo(string_para_adicionar=(string_para_adicionar, "parâmetro", str))

        linhas = string.split("\n")

        for linha in range(1, len(linhas)):
            linhas[linha] = string_para_adicionar + linhas[linha]

```

```
return "\n".join(linhas)
```

```
[14]: def binario_para_decimal(bits):  
    bits = verifica_tipo(bits=(bits, "parâmetro", np.ndarray))  
  
    return int("".join(bits.astype(np.int_).astype(str)), base=2) - 1
```

```
[15]: def decimal_para_binario(numero, numero_de_bits):  
    numero = verifica_tipo(numero=(numero, "parâmetro", t.SupportsInt))  
    digitos_binarios = list(bin(numero + 1)[2:].zfill(numero_de_bits))  
  
    verifica_comprimento_binario_igual_a(numero=(digitos_binarios, "parâmetro"),  
→ numero_de_bits=(numero_de_bits, "parâmetro"))  
  
    return np.array(digitos_binarios, dtype=np.int_).astype(np.bool_)
```

## 2.2 A classe Tabuleiro

Uma vez definidas as funções auxiliares, definamos agora a classe Tabuleiro. Alguns recursos permitidos por esta implementação incluem

- suporte às notações binária e decimal (i.e., inteira), com possibilidade de alteração entre notações após a inicialização do objeto;
- suporte a operadores e funções internas ao python, como `__str__`, `__repr__`, `__lt__` (<), `__eq__` (=), `__gt__` (>) e outras;
- verificação de ataques (i.e., a função objetivo definida para este problema);
- homogeneização das heurísticas adotadas na inicialização (no caso, não aceitar rainhas na mesma linha ou na mesma coluna);
- suporte a diferentes tamanhos de tabuleiro e números de rainhas (o que quer dizer que Tabuleiro representa também a *ausência* de rainhas),

entre outros. Embora se tenha o entendimento de que alguns recursos da implementação não serão utilizados na resolução dos exercícios ora propostos (e.g., o último recurso citado), eles compõem a implementação e a tornam mais flexível, aumentando a possibilidade de reciclagem desta peça de código para outros fins.

```
[16]: class Tabuleiro:  
    def __init__(self, *, binario=False, lado_tabuleiro=8, n_rainhas=8,  
→ rainhas=None):  
        self.__lado_tabuleiro, self.__n_rainhas, self.__rainhas, self.__valor =  
→ None, None, None, np.inf  
  
        self.binario = binario  
        self.lado_tabuleiro = lado_tabuleiro  
        self.n_rainhas = n_rainhas  
  
        if rainhas is not None:  
            self.rainhas = rainhas
```

```

@property
def binario(self):
    return self.__binario

@binario.setter
def binario(self, novo_binario):
    novo_binario = verifica_tipo(binario=(novo_binario, "atributo", (bool,
→np.bool_)))

    self.__binario = novo_binario

    if self.__rainhas is not None:
        if novo_binario is True:
            numero_de_casas = np.ceil(np.log2(self.lado_tabuleiro + 1)).
→astype(np.int_)
            self.rainhas = [decimal_para_binario(posicao, numero_de_casas)
→for posicao in self.rainhas]
        else:
            self.rainhas = [binario_para_decimal(posicao) for posicao in
→self.rainhas]

@property
def lado_tabuleiro(self):
    return self.__lado_tabuleiro

@lado_tabuleiro.setter
def lado_tabuleiro(self, novo_lado_tabuleiro):
    novo_lado_tabuleiro = verifica_tipo(lado_tabuleiro=(novo_lado_tabuleiro,
→"atributo", t.SupportsInt))

    verifica_nao_negatividade(lado_tabuleiro=(novo_lado_tabuleiro,
→"atributo"))
    verifica_maior_ou_igual_a(lado_tabuleiro=(novo_lado_tabuleiro,
→"atributo"), n_rainhas=(self.n_rainhas, "atributo"))

    self.__lado_tabuleiro = novo_lado_tabuleiro

@property
def n_rainhas(self):
    return self.__n_rainhas

@n_rainhas.setter
def n_rainhas(self, novo_n_rainhas):
    novo_n_rainhas = verifica_tipo(n_rainhas=(novo_n_rainhas, "atributo", t.
→SupportsInt))

```



```

        verifica_nao_negatividade(n_rainhas=(novo_n_rainhas, "atributo"))
        verifica_menor_ou_igual_a(n_rainhas=(novo_n_rainhas, "atributo"),
→lado_tabuleiro=(self.lado_tabuleiro, "atributo"))

        self.__n_rainhas = novo_n_rainhas

        self.aleatoriza_rainhas()

    def aleatoriza_rainhas(self):
        posicoes = np.random.choice(self.lado_tabuleiro, self.n_rainhas,
→replace=False)

        if self.binario is True:
            digitos_necessarios = np.ceil(np.log2(self.lado_tabuleiro + 1)).
→astype(np.int_)
            rainhas = np.empty((self.lado_tabuleiro, digitos_necessarios),
→dtype=np.bool_)
            rainhas[:self.n_rainhas, :] = [decimal_para_binario(posicao,
→digitos_necessarios) for posicao in posicoes]
            rainhas[self.n_rainhas:, :] = np.False_
        else:
            rainhas = np.empty(self.lado_tabuleiro, dtype=np.int_)

            rainhas[:self.n_rainhas] = posicoes
            rainhas[self.n_rainhas:] = -1

        self.rainhas = rainhas

    @property
    def rainhas(self):
        return self.__rainhas.copy() if self.__rainhas is not None else None

    @rainhas.setter
    def rainhas(self, novo_rainhas):
        novo_rainhas = verifica_tipo(rainhas=(novo_rainhas, "atributo", np.
→ndarray))

        verifica_comprimento_menor_ou_igual_a(rainhas=(novo_rainhas,
→"atributo"), lado_tabuleiro=(self.lado_tabuleiro, "atributo"))

        if self.binario is True:
            novo_rainhas = verifica_dtype(rainhas=(novo_rainhas, "atributo", np.
→bool_))

        verifica_ndim(rainhas=(novo_rainhas, "atributo", 2))

```

```

        if novo_rainhas.shape[0] < self.lado_tabuleiro:
            diferenca = self.lado_tabuleiro - novo_rainhas.shape[0]
            novo_rainhas = np.concatenate((novo_rainhas, diferenca * [[np.
→False_, np.False_, np.False_]]))
        else:
            novo_rainhas = verifica_dtype(rainhas=(novo_rainhas, "atributo", np.
→int_))

            verifica_ndim(rainhas=(novo_rainhas, "atributo", 1))

        if novo_rainhas.shape[0] < self.lado_tabuleiro:
            diferenca = self.lado_tabuleiro - novo_rainhas.shape[0]
            novo_rainhas = np.append(novo_rainhas, diferenca * [-1])

    self.__rainhas = novo_rainhas

    self.calcula_valor()

def calcula_valor(self):
    ataques = 0

    for indice_rainha in range(self.n_rainhas):
        for indice_prox_rainha in range(indice_rainha + 1, self.n_rainhas):
            ataques += self.ha_ataque(indice_rainha, indice_prox_rainha)

    self.__valor = ataques

@property
def valor(self):
    return self.__valor

def ha_ataque(self, indice_a, indice_b):
    if self.binario is True:
        iterador = (binario_para_decimal(posicao) for posicao in self.
→rainhas)
        rainhas = np.fromiter(iterador, np.int_, self.n_rainhas)
    else:
        rainhas = self.rainhas

    indice_a = verifica_tipo(indice_a=(indice_a, "parâmetro", t.SupportsInt))
    indice_b = verifica_tipo(indice_b=(indice_b, "parâmetro", t.SupportsInt))

    ha_ataque_diagonal = (indice_b - indice_a) == abs(rainhas[indice_b] -
→rainhas[indice_a])

    return ha_ataque_diagonal

```

```

@property
def ha_rainhas_na_mesma_linha(self):
    if self.binario is True:
        iterador = (binario_para_decimal(posicao) for posicao in self.
→rainhas)
        rainhas = np.fromiter(iterador, np.int_, self.n_rainhas)
    else:
        rainhas = self.rainhas

    rainhas = rainhas[rainhas != -1]

    return True if rainhas.shape[0] != np.unique(rainhas).shape[0] else False

def __sub__(self, outro):
    verifica_tipo_operador('-', outro, Tabuleiro)

    return self.valor - outro.valor

def __lt__(self, outro):
    verifica_tipo_operador('<', outro, Tabuleiro)

    return self.valor < outro.valor

def __gt__(self, outro):
    verifica_tipo_operador('>', outro, Tabuleiro)

    return self.valor > outro.valor

def __eq__(self, outro):
    verifica_tipo_operador('==', outro, Tabuleiro)

    return self.valor == outro.valor

def __repr__(self):
    if self.binario is False:
        representacao_rainhas = str(self.rainhas)
        tipo = "Decimal"
    else:
        representacao_rainhas = caixinha.ajusta_indentacao(str(self.
→rainhas), 12 * ' ')
        tipo = "Binário"

    return f"[Tabuleiro] {representacao_rainhas} | {self.valor} ataques |_
→{tipo}"

def __str__(self):

```

```

        if self.binario is False:
            return str(self.rainhas.tolist())
        else:
            representacao = []

            for rainha in self.rainhas:
                string = ""

                for bit in rainha:
                    if bit is np.True_:
                        string += "1"
                    else:
                        string += "0"

                representacao.append(string)

            return str(representacao).replace("'", "")

    def __len__(self):
        return self.n_rainhas

```

## 2.3 A função-objetivo

A função objetivo utilizada, retornada por `Tabuleiro.valor` e calculada em `Tabuleiro.calcula_valor()`, verifica a existência de ataques de rainhas no tabuleiro na diagonal, uma vez que as heurísticas adotadas não permitem rainhas na mesma coluna ou linha. Ela é utilizada por ambos os algoritmos solicitados pela atividade, que serão descritos mais adiante neste documento. Sua implementação é mostrada abaixo:

```

def calcula_valor(self):
    ataques = 0

    for indice_rainha in range(self.n_rainhas):
        for indice_prox_rainha in range(indice_rainha + 1, self.n_rainhas):
            ataques += self.ha_ataque(indice_rainha, indice_prox_rainha)

    self.__valor = ataques

def ha_ataque(self, indice_a, indice_b):
    if self.binario is True:
        iterador = (binario_para_decimal(posicao) for posicao in self.rainhas)
        rainhas = np.fromiter(iterador, np.int_, self.n_rainhas)
    else:
        rainhas = self.rainhas

    indice_a = verifica_tipo(indice_a=(indice_a, "parâmetro", t.SupportsInt))
    indice_b = verifica_tipo(indice_b=(indice_b, "parâmetro", t.SupportsInt))

```

```

ha_ataque_diagonal = (indice_b - indice_a) == abs(rainhas[indice_b] -
                                                    rainhas[indice_a])

return ha_ataque_diagonal

```

Dado que o tabuleiro é representado por um vetor com  $n$  elementos tal que o valor de cada elemento da posição  $i$  representa a linha onde a rainha se encontra nessa mesma coluna  $i$ , percorre-se esse vetor varrendo para cada elemento da posição `indice_a` os elementos seguintes que estão na posição `indice_b`.

Para detectar ataques diagonais, verifica-se se a diferença entre o `indice_b` e `indice_a` é igual ao valor absoluto da diferença entre o item da posição `indice_b` e o item da posição `indice_a`. Esta regra foi utilizada pois observou-se que os elementos da diagonal estão uma (ou mais) coluna(s) na frente e uma (ou mais) linha(s) abaixo do elemento anterior a eles na diagonal (exceto pelo primeiro elemento, que não possui elemento anterior). Dessa forma, a diferença entre os valores da coluna e a diferença entre os valores da linha dos elementos  $a$  e  $b$  será igual quando eles estiverem na mesma diagonal. Por isso a diferença entre as colunas, representados por `indice_b` e `indice_a`, e a diferença entre as linhas, que são os valores armazenados nos vetores, representados por `rainhas[indice_b]` e `rainhas[indice_a]`, são calculadas.

Com isso, caso algum ataque seja detectado, soma-se 1 ao valor do tabuleiro.

## 2.4 As soluções

Uma solução, da forma como está codificada na classe `Tabuleiro`, pode ser descrita de duas formas: binária e decimal (i.e., inteira). Cada solução consiste, sucintamente, em um vetor em que cada número corresponde à linha em que a rainha se encontra em uma dada coluna.

Uma vez que a implementação da classe `Tabuleiro` foi feita de forma a suportar diferentes números de rainhas e tamanhos do lado do tabuleiro (que é sempre quadrado), tem-se que há um valor que representa a ausência de rainha em uma dada coluna:

- **Na notação binária**, o valor 0 (representado aqui, em notação decimal, por -1) representa a ausência de rainha em uma dada coluna; e
- **Na notação decimal**, o valor -1 (representado aqui, em notação binária, por 0) representa a ausência de rainha em uma dada coluna.

Isso significa, em suma, que há um deslocamento em uma unidade na notação binária em relação à notação decimal. Para fins ilustrativos, analisemos a seguinte solução para um tabuleiro quadrado de lado 8 contendo 8 rainhas, tal que os índices possíveis do vetor variem de 0 a 7:

[0011, 0110, 1000, 0001, 0100, 0111, 0101, 0010]

A sua versão decimal seria:

[0002, 0005, 0007, 0000, 0003, 0006, 0004, 0001]

Se ignorássemos esse deslocamento, erros ocorreriam a nível de implementação, uma vez que o Python, assim como Java e outras linguagens, possui vetores *zero-based*. Contudo, em termos de tabuleiro, isso não seria problemático. Se ignorássemos o deslocamento, teríamos a seguinte solução no mundo real:

[0003, 0006, 0008, 0001, 0004, 0007, 0005, 0002]

Logo, teríamos rainhas:

- Na primeira coluna e terceira linha;
- Na segunda coluna e sexta linha;
- Na terceira coluna e oitava linha;
- Na quarta coluna e primeira linha;
- Na quinta coluna e quarta linha;
- Na sexta coluna e sétima linha;
- Na sétima coluna e quinta linha; e
- Na oitava coluna e segunda linha.

Note que, neste exemplo, não há colunas nem linhas sem rainhas, i.e., todas as colunas possuem exatamente uma rainha.

### 3 Outras definições globais

Antes de seguirmos às resoluções para os problemas da atividade, definamos aqui alguns aspectos globalmente aplicados a ambos os algoritmos.

#### 3.1 Heurísticas adotadas

Como conhecimento prévio estipulado pelo projetista, foi estipulado que as rainhas não podem estar em uma mesma coluna ou linha. Quaisquer soluções que não adiram a esses requisitos são descartadas.

#### 3.2 Critérios de parada adotados

Dois critérios de parada são adotados por ambos os algoritmos:

1. quando o número máximo de iterações for atingido; e
2. quando uma solução ótima for encontrada.

## 4 Problemas

Com o auxílio das definições-suporte feitas acima, pode-se seguir para as que dizem respeito aos algoritmos. Antes de fazê-lo, contudo, é necessário que definamos uma função para os experimentos que serão conduzidos, a fim de se produzirem as tabelas, gráficos e informações estipuladas em algumas alíneas desta atividade.

```
[17]: def executa_experimento(funcao, n_execucoes=50, verboso=True):  
    n_execucoes = verifica_tipo(n_execucoes=(n_execucoes, "parâmetro", t.  
    ↳ SupportsInt))  
    verboso = verifica_tipo(verboso=(verboso, "parâmetro", (bool, np.bool_)))  
  
    melhores_solucoes = np.empty((n_execucoes, 1), dtype=np.object_)  
    melhores_valores = np.empty((n_execucoes, 1), dtype=np.int_)
```

```

tempos_gastos = np.empty((n_execucoes, 1), dtype=np.float_)
iteracoes_gastas = np.empty((n_execucoes, 1), dtype=np.int_)

for e in range(n_execucoes):
    if verboso is True:
        print(f"Execução {e + 1}...", end="\r")

        tempos_gastos[e, 0] = timer()
        melhores_solucoes[e, 0], iteracoes_gastas[e, 0] = funcao()
        tempos_gastos[e, 0] = timer() - tempos_gastos[e, 0]
        melhores_valores[e, 0] = melhores_solucoes[e, 0].valor
        melhores_solucoes[e, 0] = str(melhores_solucoes[e, 0])

    tabela = np.concatenate((melhores_solucoes, melhores_valores, tempos_gastos,
→iteracoes_gastas), axis=1)
    tabela = pd.DataFrame(tabela, columns=["Solução", "Função-objetivo", "Tempo",
→(seg.)", "Iterações"]).infer_objects()

    figura, eixos = plt.subplots(1, 2, figsize=(10, 5))

    # figura.tight_layout()
    eixos[0].set_xlim(1, 50)
    eixos[1].set_xlim(1, 50)

    figura.suptitle(f"Dados da execução do algoritmo {str(funcao).split(' '
→')[1]}", y=1.05)

    eixos[0].set_title("Iterações por execução")
    eixos[1].set_title("Tempo por execução (seg.)")

    sns.lineplot(x=range(1, n_execucoes + 1), y=iteracoes_gastas.ravel(),
→ax=eixos[0])
    sns.lineplot(x=range(1, n_execucoes + 1), y=tempos_gastos.ravel(),
→ax=eixos[1])

    if verboso is True:
        print(f"Execução concluída.")

    return tabela, figura

```

Por fim, definamos aqui uma função que desempenhe a análise de correlação linear que faremos mais adiante.

```

[18]: def verifica_correlacao(funcao, tabela):
        tabela_ordenada = tabela.sort_values(["Iterações", "Tempo (seg.)"])
        pearson = stats.pearsonr(tabela['Tempo (seg.)'].to_numpy(),
→tabela['Iterações'].to_numpy())[0]

```

```

figura_correlacao, eixo_correlacao = plt.subplots(1, 1)

figura_correlacao.tight_layout()
figura_correlacao.suptitle(f"{str(funcao).split(' ')[1]}: Iterações vs. ↳
↳Tempo (seg.)", y=1.05)

sns.lineplot(x=tabela_ordenada["Iterações"].to_numpy().ravel(), ↳
↳y=tabela_ordenada["Tempo (seg.)"].to_numpy().ravel(), ax=eixo_correlacao)

return figura_correlacao, pearson

```

## 4.1 Stochastic Hill Climbing: Simulated Annealing

Para resolver a primeira questão, que diz respeito ao *Stochastic Hill Climbing* (SHC), optamos por implementar o *Simulated Annealing* (SA).

### 4.1.1 Implementação do SA

Antes de seguirmos à resolução para as alíneas referentes a este algoritmo, definamos algumas funções.

**Funções auxiliares ao SA** Uma função auxiliar necessária precisa produzir um vetor com as temperaturas a serem iteradas pelo algoritmo. Nesta implementação, utilizou-se o decaimento linear da temperatura.

```

[19]: def gera_temperaturas(temperatura_inicial, variacao):
        temperatura_inicial = ↳
        ↳verifica_tipo(temperatura_inicial=(temperatura_inicial, "parâmetro", t.
        ↳SupportsFloat))
        variacao = verifica_tipo(variacao=(variacao, "parâmetro", t.SupportsFloat))

        n_iteracoes = np.round(np.floor(temperatura_inicial / variacao))
        temperaturas = temperatura_inicial - variacao * np.arange(n_iteracoes)

        return temperaturas

```

Por fim, uma segunda função auxiliar é necessária para definir se o algoritmo deve ir para uma solução pior (i.e., uma solução com mais ataques).

```

[20]: def tabuleiro_deve_mudar(temperatura, variacao):
        temperatura = verifica_tipo(temperatura=(temperatura, "parâmetro", t.
        ↳SupportsFloat))
        variacao = verifica_tipo(variacao=(variacao, "parâmetro", t.SupportsFloat))
        e = float(np.e)

        exp = d.Decimal(d.Decimal(e) ** (d.Decimal(variacao) / d.
        ↳Decimal(temperatura)))

```



```
return np.random.uniform() < exp
```

**Algoritmo principal SA** Com base nas definições acima, definamos agora o algoritmo principal.

```
[21]: def recozimento_simulado(temperatura_inicial=100, variacao=.01, binario=False,
    ↳verboso=False):
    temperatura_inicial =
    ↳verifica_tipo(temperatura_inicial=(temperatura_inicial, "parâmetro", t.
    ↳SupportsFloat))
    variacao = verifica_tipo(variacao=(variacao, "parâmetro", t.SupportsFloat))
    verboso = verifica_tipo(verboso=(verboso, "parâmetro", (bool, np.bool_)))

    verifica_menor_ou_igual_a(variacao=(variacao, "parâmetro"),
    ↳temperatura_inicial=(temperatura_inicial, "parâmetro"))

    tabuleiro_atual = Tabuleiro(binario=binario)
    temperaturas = gera_temperaturas(temperatura_inicial, variacao)

    if verboso is True:
        print(f"[Iteração -1] {tabuleiro_atual}")

    if tabuleiro_atual.valor == 0:
        return tabuleiro_atual, 0

    for iteracao, temperatura in enumerate(temperaturas):
        tabuleiro_seguinte = Tabuleiro(binario=binario)

        if tabuleiro_seguinte.valor == 0:
            return tabuleiro_seguinte, iteracao + 1

        variacao = tabuleiro_atual - tabuleiro_seguinte

        if variacao > 0:
            tabuleiro_atual = tabuleiro_seguinte
        elif tabuleiro_deve_mudar(temperatura, variacao):
            tabuleiro_atual = tabuleiro_seguinte

        if verboso is True:
            print(f"[Iteração {iteracao}] {tabuleiro_atual} [Temperatura {np.
    ↳round(temperatura, 3)}]")

    if verboso is True:
        print("A temperatura chegou a zero.")

    return tabuleiro_atual
```

Esta implementação foi realizada nas linhas do pseudo-código exposto na aula para este algoritmo, utilizando a codificação decimal (i.e., inteira). Nota-se que, graças ao suporte tanto à notação decimal quanto binária por parte da classe `Tabuleiro` e à simplicidade deste algoritmo, pode-se utilizar também a decimal ao regular o parâmetro binário da função `recozimento_simulado()`.

A cada rodada, a temperatura sofre um decaimento seguindo alguma função (neste caso, utilizou-se uma função de decaimento linear cujo coeficiente angular é passado em `variacao`) e um novo posicionamento das rainhas é produzido por meio da criação de uma nova instância da classe `Tabuleiro`. Caso essa solução seja melhor do que a atual (i.e., tenha menos ataques, ou um atributo valor menor), o algoritmo a toma como a solução atual. Caso contrário, a função `tabuleiro_deve_mudar()` é chamada passando a temperatura e o coeficiente angular da função de decaimento como parâmetros, de forma que se obtenha uma resposta booleana à seguinte pergunta: “o algoritmo deve escolher uma solução pior?” A depender da resposta, o algoritmo pode tomar (ou não) a solução pior como solução atual e seguir para as próximas iterações.

O algoritmo cessa sua execução quando a temperatura chega a 0, e retorna a solução atual.

#### 4.1.2 Heurísticas e critérios de parada do SA

As heurísticas e critérios de parada adotados aqui são os globalmente definidos (rainhas não podem estar na mesma linha nem na mesma coluna). O mesmo vale para os critérios de parada: o algoritmo finaliza sua execução quando

1. a temperatura chega a zero (i.e., o vetor de *arrays* de temperaturas, cujo tamanho depende da temperatura inicial e variação e define o número de iterações, é integralmente percorrido); e
2. o número de ataques chega a zero (i.e., `tabuleiro_atual.valor == 0` ou `tabuleiro_seguinte.valor == 0`).

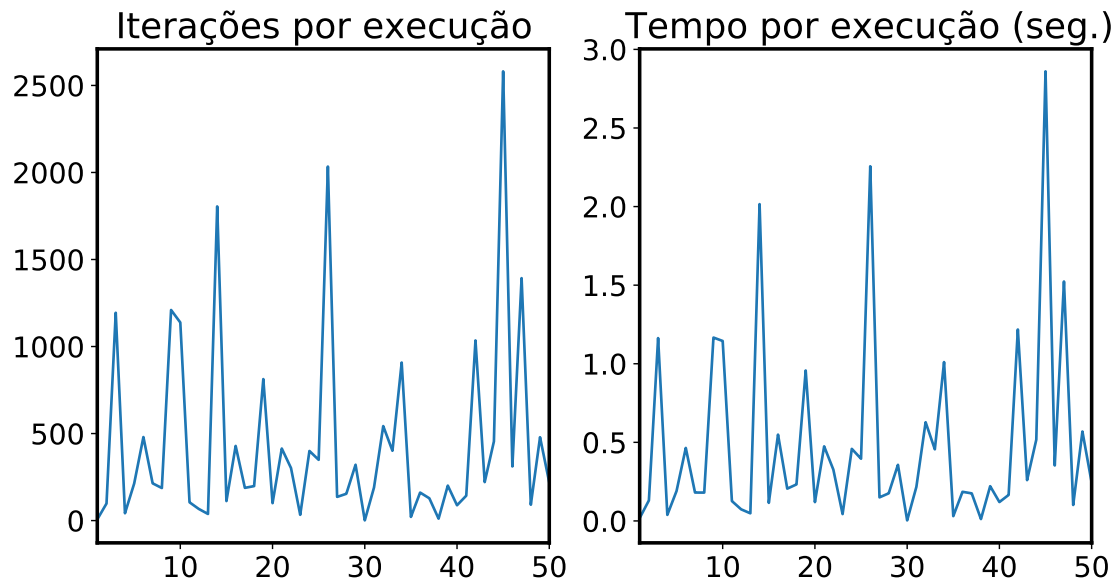
#### 4.1.3 Resultados do SA

Pode-se obter os resultados pedidos através da função `executa_experimento()`, passando `recozimento_simulado()` pelo parâmetro `funcao`.

```
[22]: tabelaSA, figuraSA = executa_experimento(recozimento_simulado)
```

Execução concluída.

## Dados da execução do algoritmo recozimento\_simulado

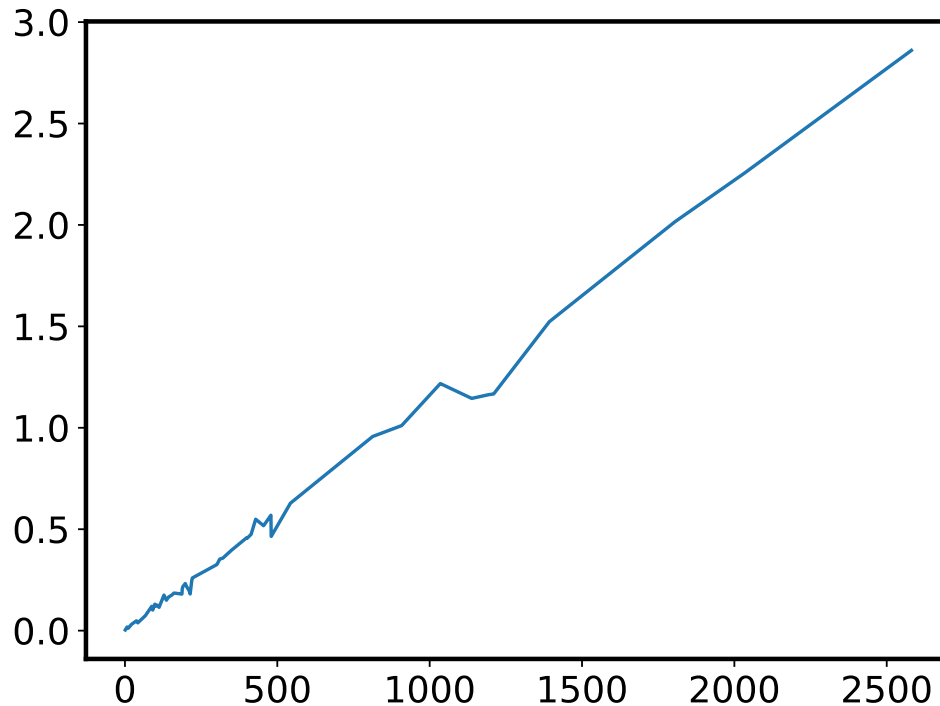


Neste primeiro gráfico, temos o detalhamento da execução do algoritmo `recozimento_simulado()`. À esquerda, tem-se o número de iterações por execução; à direita, o tempo por execução em segundos. À primeira vista, nota-se a semelhança no comportamento entre os dois plots e a diferença na escala adotada no eixo das ordenadas pelos dois plots, o que dá a entender que o tempo cresce linearmente em função do número de iterações. Vamos tentar confirmar essa tendência dos resultados produzidos de duas formas: visualmente e numericamente.

Em termos visuais, pode-se verificar a presença de correlação linear ao produzir um plot cujo eixo das abscissas representa as iterações e eixo das ordenadas, o tempo, produzido sobre a tabela de resultados ordenada em função do número de iterações e tomando o tempo como critério de desempate.

```
[23]: figuraSAcorrelacao, pearsonSA = verifica_correlacao(recozimento_simulado, t  
    ->tabelaSA)
```

## recozimento\_simulado: Iterações vs. Tempo (seg.)



Em termos numéricos, pode-se verificar a presença de correlação linear ao utilizar o coeficiente de correlação estatística Pearson, que mede a presença de correlação linear em duas distribuições de valores. Quando o valor se aproxima de 1, tem-se uma forte correlação positiva; quando o valor se aproxima de -1, tem-se uma forte correlação negativa; e quando o valor se aproxima de 0, não há correlação linear.

```
[24]: print(f"Coeficiente de correlação Pearson dos resultados de recozimento_simulado:  
      → {np.round(pearsonSA, 3)}")
```

Coeficiente de correlação Pearson dos resultados de recozimento\_simulado: 0.997

Tem-se constatada, portanto, uma forte correlação linear nos resultados apresentados. Vejamos algumas entradas da tabela de resultados, em que algumas soluções produzidas pelo algoritmo podem ser vistas.

```
[25]: tabelaSA
```

```
[25]:
```

	Solução	Função-objetivo	Tempo (seg.)	Iterações
0	[5, 1, 6, 0, 3, 7, 4, 2]	0	0.018009	7
1	[7, 3, 0, 2, 5, 1, 6, 4]	0	0.130405	98
2	[5, 2, 0, 7, 3, 1, 6, 4]	0	1.163332	1194
3	[5, 3, 1, 7, 4, 6, 0, 2]	0	0.038340	42
4	[3, 1, 4, 7, 5, 0, 2, 6]	0	0.190607	213
5	[3, 7, 0, 2, 5, 1, 6, 4]	0	0.464528	480
6	[7, 1, 3, 0, 6, 4, 2, 5]	0	0.180725	214
7	[3, 1, 7, 5, 0, 2, 4, 6]	0	0.180212	187
8	[5, 2, 4, 7, 0, 3, 1, 6]	0	1.166690	1210
9	[3, 0, 4, 7, 5, 2, 6, 1]	0	1.144864	1138
10	[5, 2, 6, 1, 3, 7, 0, 4]	0	0.126367	105
11	[3, 7, 0, 4, 6, 1, 5, 2]	0	0.073845	67
12	[2, 6, 1, 7, 5, 3, 0, 4]	0	0.048458	38
13	[6, 1, 3, 0, 7, 4, 2, 5]	0	2.015708	1805
14	[1, 7, 5, 0, 2, 4, 6, 3]	0	0.114864	112
15	[4, 0, 7, 5, 2, 6, 1, 3]	0	0.549206	429
16	[4, 2, 7, 3, 6, 0, 5, 1]	0	0.205249	188
17	[4, 6, 0, 2, 7, 5, 3, 1]	0	0.232352	198
18	[3, 6, 4, 1, 5, 0, 2, 7]	0	0.957179	813
19	[6, 4, 2, 0, 5, 7, 1, 3]	0	0.119675	100
20	[3, 0, 4, 7, 5, 2, 6, 1]	0	0.474622	414
21	[4, 6, 1, 5, 2, 0, 7, 3]	0	0.326074	302
22	[2, 0, 6, 4, 7, 1, 3, 5]	0	0.043434	33
23	[2, 5, 7, 1, 3, 0, 6, 4]	0	0.458578	400
24	[6, 1, 5, 2, 0, 3, 7, 4]	0	0.395698	349
25	[3, 6, 2, 7, 1, 4, 0, 5]	0	2.256433	2034
26	[3, 0, 4, 7, 1, 6, 2, 5]	0	0.149926	136
27	[6, 1, 3, 0, 7, 4, 2, 5]	0	0.175611	154
28	[3, 0, 4, 7, 1, 6, 2, 5]	0	0.357115	321
29	[3, 1, 4, 7, 5, 0, 2, 6]	0	0.003323	1
30	[5, 1, 6, 0, 2, 4, 7, 3]	0	0.216795	190
31	[2, 5, 1, 6, 0, 3, 7, 4]	0	0.628183	543
32	[2, 5, 3, 1, 7, 4, 6, 0]	0	0.455530	401
33	[1, 5, 7, 2, 0, 3, 6, 4]	0	1.010713	908
34	[1, 7, 5, 0, 2, 4, 6, 3]	0	0.030360	21
35	[0, 4, 7, 5, 2, 6, 1, 3]	0	0.185371	161
36	[2, 5, 7, 0, 3, 6, 4, 1]	0	0.175295	128
37	[3, 1, 7, 4, 6, 0, 2, 5]	0	0.011832	11
38	[4, 6, 1, 3, 7, 0, 2, 5]	0	0.221057	201
39	[2, 6, 1, 7, 4, 0, 3, 5]	0	0.120031	88
40	[1, 7, 5, 0, 2, 4, 6, 3]	0	0.165382	143
41	[3, 6, 4, 2, 0, 5, 7, 1]	0	1.217966	1035
42	[4, 0, 7, 3, 1, 6, 2, 5]	0	0.259439	221
43	[3, 6, 2, 7, 1, 4, 0, 5]	0	0.517157	455
44	[5, 2, 4, 7, 0, 3, 1, 6]	0	2.860380	2581
45	[2, 5, 1, 6, 4, 0, 7, 3]	0	0.352824	311
46	[4, 2, 0, 5, 7, 1, 3, 6]	0	1.523361	1393
47	[3, 0, 4, 7, 5, 2, 6, 1]	0	0.101384	91
48	[2, 5, 7, 0, 4, 6, 1, 3]	0	0.568954	479
49	[4, 1, 3, 5, 7, 2, 0, 6]	0	0.244477	219

Por fim, vejamos algumas estatísticas obtidas a partir da tabela de resultados do SA.

```
[26]: print(f"Média e desvio padrão da função-objetivo: {tabelaSA['Função-objetivo']}.
      →mean()} ± {tabelaSA['Função-objetivo'].std()} ataques\n"
      f"Média e desvio padrão do tempo gasto: {np.round(tabelaSA['Tempo (seg.)']
      →mean(), 3)} ± {np.round(tabelaSA['Tempo (seg.)'].std(), 3)} segundos\n"
      f"Média e desvio padrão das iterações gastas: {np.
      →round(tabelaSA['Iterações'].mean(), 3)} ± {np.round(tabelaSA['Iterações']
      →std(), 3)}")
```

Média e desvio padrão da função-objetivo: 0.0 ± 0.0 ataques  
 Média e desvio padrão do tempo gasto: 0.493 ± 0.611 segundos  
 Média e desvio padrão das iterações gastas: 447.24 ± 559.441

Nota-se que temos um desvio padrão no tempo e no número de iterações praticamente igual, respectivamente, ao tempo e ao número de iterações médios, o que demonstra uma significativa oscilação no número de iterações e no tempo entre execuções. Por um outro lado, tem-se que todas as execuções conseguiram encontrar uma solução ótima antes da temperatura chegar a zero.

## 4.2 Meta-heuristic: Genetic Algorithm

Para resolver a segunda questão, realizamos a implementação da meta-heurística *Genetic Algorithm* (GA).

### 4.2.1 Implementação do GA

Antes de seguirmos à resolução para as alíneas referentes a este algoritmo, definamos algumas funções.

**Funções auxiliares ao GA** Uma definição é necessária para auxiliar na exibição da evolução do algoritmo quando `verboso = True`.

```
[27]: def gera_contagem_populacao(populacao):
      ataques = [tabuleiro.valor for tabuleiro in populacao]

      colunas = np.unique(ataques)
      indices = ["Contagem"]

      contagem_de_valores = {ataque: 0 for ataque in colunas}

      for ataque in ataques:
          contagem_de_valores[ataque] += 1

      data_frame = pd.DataFrame(contagem_de_valores, columns=colunas,
      →index=indices)
      data_frame.columns.names = ["Ataques"]

      return data_frame
```

Em adição, uma função é requerida para inicializar a população.

```
[28]: def inicializa_populacao(tamanho_populacao):
    populacao = [Tabuleiro(binario=True) for _ in range(tamanho_populacao)]

    return sorted(populacao)
```

Em seguida, definições que aplicam os operadores evolucionários são necessárias.

```
[29]: def selecao_dos_pais(populacao, tamanho_ringue):
    nova_populacao, tamanho_populacao = [], len(populacao)

    for _ in range(tamanho_populacao):
        lutadores = r.sample(populacao, tamanho_ringue)
        lutadores = sorted(lutadores)

        nova_populacao.append(lutadores[0])

    return nova_populacao
```

```
[30]: def cruzamento(populacao, taxa_crossover):
    nova_populacao, tamanho_populacao = [], len(populacao)
    n_rodadas = np.ceil(tamanho_populacao / 2).astype(np.int_)

    for _ in range(n_rodadas):
        pai1, pai2 = r.sample(populacao, 2)
        pai1, pai2 = pai1.rainhas, pai2.rainhas

        if np.random.uniform() <= taxa_crossover:
            corte = r.randint(1, len(pai1) - 1)

            filho1 = np.concatenate((pai1[:corte], pai2[corte:]))
            filho2 = np.concatenate((pai2[:corte], pai1[corte:]))
        else:
            filho1, filho2 = pai1, pai2

        filho1, filho2 = Tabuleiro(binario=True, rainhas=filho1),
        ↳ Tabuleiro(binario=True, rainhas=filho2)

        if tamanho_populacao - len(nova_populacao) >= 2:
            nova_populacao.extend([filho1, filho2])
        else:
            nova_populacao.append(filho1 if np.random.uniform() <= .5 else
            ↳ filho2)

    return nova_populacao
```

```
[31]: def mutacao(populacao, taxa_mutacao):
    nova_populacao, tamanho_populacao = [], len(populacao)
```

```

for i in range(tamanho_populacao):
    rainhas = populacao[i].rainhas

    for bit in range(rainhas.ravel().shape[0]):
        if np.random.uniform() <= taxa_mutacao:
            rainhas.ravel()[bit] = not rainhas.ravel()[bit]

    novo_tabuleiro = Tabuleiro(binario=True, rainhas=rainhas)

    nova_populacao.append(novo_tabuleiro)

return nova_populacao

```

```

[32]: def seleciona_sobreviventes(populacao, nova_populacao):
    sobreviventes = sorted(populacao + nova_populacao)

    return sobreviventes[:len(populacao)]

```

Em seguida, uma função que assegura que os filhos estão respeitando as heurísticas estipuladas (nenhuma rainha na mesma linha) é definida.

```

[33]: def gera_filhos_validos(pais, taxa_crossover, taxa_mutacao):
    filhos = []

    embrioes = cruzamento(pais, taxa_crossover)
    fetos = mutacao(embrioes, taxa_mutacao)

    for feto in fetos:
        if feto.ha_rainhas_na_mesma_linha is False:
            filhos.append(feto)

    return filhos

```

Por fim, uma solução que gera a nova população é definida.

```

[34]: def gera_nova_populacao(populacao, taxa_crossover, taxa_mutacao, tamanho_ringue):
    pais = selecao_dos_pais(populacao, tamanho_ringue)
    filhos = gera_filhos_validos(pais, taxa_crossover, taxa_mutacao)

    return sorted(filhos)

```

**Algoritmo principal GA** Com base nas definições acima, definamos agora o algoritmo principal.

```

[35]:

```



```

def algoritmo_genetico(tamanho_populacao=20, max_iteracoes=1000, taxa_mutacao=.
→03, taxa_crossover=.8, tamanho_ringue=3, verboso=False):
    tamanho_populacao = verifica_tipo(tamanho_populacao=(tamanho_populacao,
→"parâmetro", t.SupportsInt))
    max_iteracoes = verifica_tipo(max_iteracoes=(max_iteracoes, "parâmetro", t.
→SupportsInt))
    taxa_mutacao = verifica_tipo(taxa_mutacao=(taxa_mutacao, "parâmetro", t.
→SupportsFloat))
    taxa_crossover = verifica_tipo(taxa_crossover=(taxa_crossover, "parâmetro",
→t.SupportsFloat))
    tamanho_ringue = verifica_tipo(tamanho_do_ringue=(tamanho_ringue,
→"parâmetro", t.SupportsInt))
    verboso = verifica_tipo(verboso=(verboso, "parâmetro", (bool, np.bool_)))

    verifica_nao_negatividade(tamanho_populacao=(tamanho_populacao,
→"parâmetro"), max_iteracoes=(max_iteracoes, "parâmetro"),
→taxa_mutacao=(taxa_mutacao, "parâmetro"), taxa_crossover=(taxa_crossover,
→"parâmetro"), tamanho_ringue=(tamanho_ringue, "parâmetro"))
    verifica_menor_ou_igual_a(taxa_mutacao=(taxa_mutacao, "parâmetro"),
→valor=(1, "valor"))
    verifica_menor_ou_igual_a(taxa_mutacao=(taxa_crossover, "parâmetro"),
→valor=(1, "valor"))
    verifica_menor_ou_igual_a(tamanho_ringue=(tamanho_ringue, "parâmetro"),
→tamanho_populacao=(tamanho_populacao, "parâmetro"))

    pais = inicializa_populacao(tamanho_populacao)
    iteracao = 0

    if verboso is True:
        print(f"[Iteração {iteracao}] População inicial: \n"
              f"{gera_contagem_populacao(pais)}", end="\n\n")

    while iteracao < max_iteracoes:
        if pais[0].valor == 0:
            if verboso is True:
                print("Uma solução ótima foi encontrada.")

            break

        filhos = gera_nova_populacao(pais, taxa_crossover, taxa_mutacao,
→tamanho_ringue)
        pais = seleciona_sobreviventes(pais, filhos)
        iteracao += 1

        if verboso is True:
            print(f"[Iteração {iteracao}]\n")

```

```

        f"Filhos: \n"
        f"{gera_contagem_populacao(filhos)}\n\n"
        f"Sobreviventes: \n"
        f"{gera_contagem_populacao(pais)}", end="\n\n")

    if iteracao >= max_iteracoes and verboso is True:
        print("O número máximo de iterações foi atingido.")

    return pais[0], iteracao

```

Nesta implementação, começa-se com a inicialização de uma população de instâncias da classe `Tabuleiro` com `binario=True`, i.e., com a notação binária em uso. A cada rodada, os operadores genéticos binários são aplicados na população, gerando filhos. Isso se dá na função `gera_nova_populacao()`.

A função `gera_nova_populacao()` não se restringe à aplicação dos operadores, no entanto: ela também descarta os filhos que possuem rainhas na mesma linha, assegurando o cumprimento das heurísticas que adotamos globalmente. Dessa forma, nenhuma solução pode ter rainhas na mesma linha nem na mesma coluna.

Por fim, a população de filhos é concatenada à de pais (gerando um vetor com o dobro do valor `tamanho_populacao`) e tem-se um processo de seleção de sobreviventes elitista, em que apenas as melhores soluções entre os pais e filhos é preservada (para isso, ordena-se o vetor `filhos+pais` e seleciona-se apenas os melhores `tamanho_populacao` elementos).

O algoritmo cessa sua execução quando atinge o número máximo de execuções ou quando o valor de alguma solução é igual a 0 (o que indica que o número de ataques do tabuleiro é igual a 0), retornando a solução atual.

#### 4.2.2 Operadores do GA

Descrevamos, agora, os operadores deste GA. É importante lembrar que, em função das estipulações da atividade proposta, apenas a implementação dos operadores de seleção dos pais, cruzamento e mutação ficaram a critério dos projetistas.

- Para a **seleção dos pais**, utilizou-se a técnica *n-Way Tournament Selection*, em que são selecionados  $n$  indivíduos aleatórios da população para “lutar” entre si, ganhando o que tiver melhor *fitness*. Adotamos  $n = 3$ . Os torneios ocorrem até que uma população de mesmo tamanho à original seja produzida (e.g., uma população de 50 indivíduos gerará uma *mating pool* de 50 possíveis pais).
- Para o **cruzamento**, utilizou-se a técnica de **cruzamento com ponto de corte aleatório**. Nessa técnica, dois indivíduos são aleatoriamente selecionados da população de pais (podendo haver repetição). Uma posição aleatória  $c$  é escolhida como ponto de corte. Quando a taxa de *crossover* não é ultrapassada, corta-se o vetor do pai 1 da posição inicial até a posição  $c$  e o vetor do pai 2 de  $c$  até a última posição. Os cortes são concatenados gerando o filho 1, e o restante dos vetores “cortados” também são concatenados para gerar o filho 2. Caso a taxa de *crossover* seja ultrapassada, os pais tornam-se os filhos. Esse processo se repete até que uma população de mesmo tamanho à original seja produzida.
- Para a **mutação**, utilizou-se a técnica de *bit flip*. Nessa técnica, há um teste para cada bit de cada solução, tal que se o número aleatório gerado for inferior ao da taxa de mutação, um bit

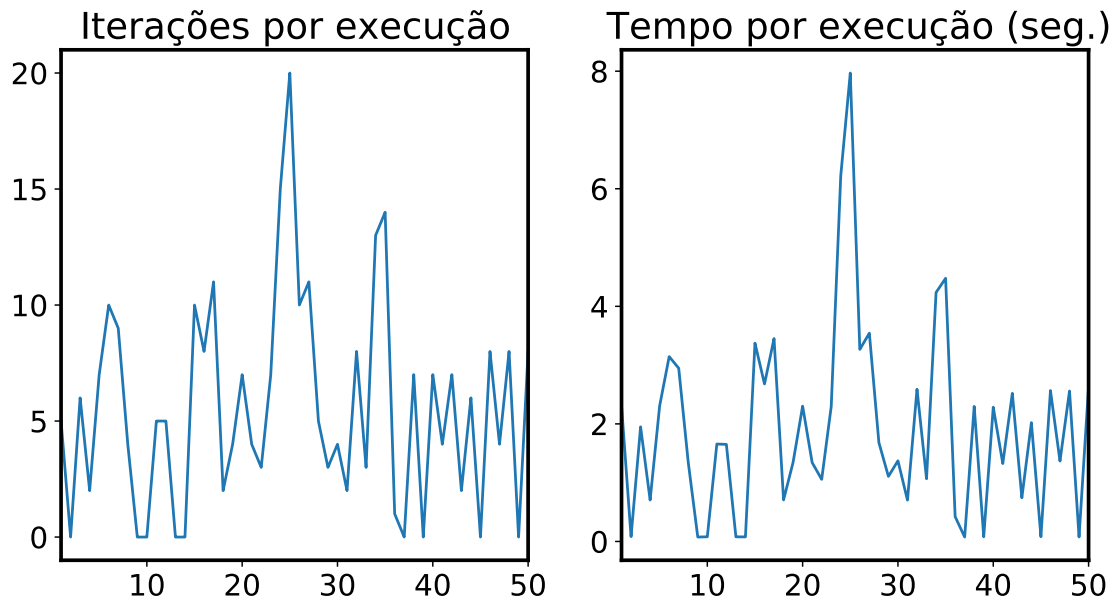
False é transformado em um bit True e vice-versa.

### 4.2.3 Resultados do GA

```
[36]: tabelaGA, figuraGA = executa_experimento(algoritmo_genetico)
```

Execução concluída.

Dados da execução do algoritmo `algoritmo_genetico`

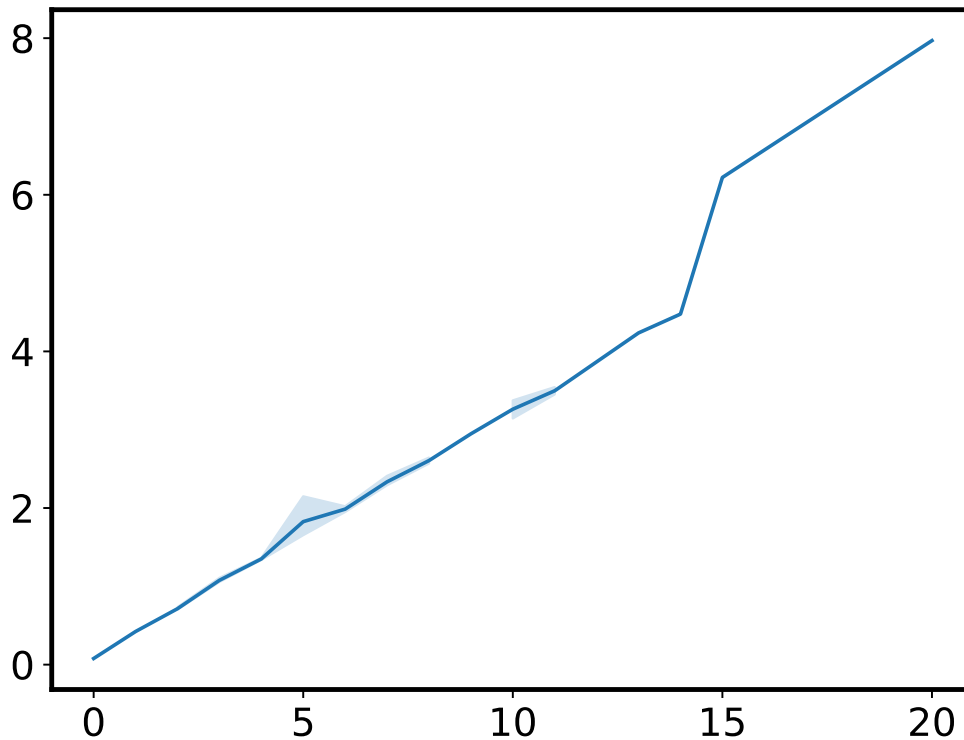


Este gráfico, similarmente ao primeiro, consiste no detalhamento da execução do algoritmo `algoritmo_genetico()`. À esquerda, tem-se o número de iterações por execução; à direita, o tempo por execução em segundos. À primeira vista, a semelhança entre o comportamento dos dois plots não é tão visível; contudo, nota-se que a escala do número de iterações e do tempo em segundos é parecida. Vamos tentar verificar a existência de correlação linear nos resultados produzidos. Fá-lo-emos, assim como ocorreu com o SA, de duas formas: visualmente e numericamente.

Em termos visuais, pode-se verificar a presença de correlação linear ao produzir um plot cujo eixo das abscissas representa as iterações e eixo das ordenadas, o tempo, produzido sobre a tabela de resultados ordenada em função do número de iterações e tomando o tempo como critério de desempate.

```
[37]: figuraGAcorrelacao, pearsonGA = verifica_correlacao(algoritmo_genetico, tabelaGA)
```

## algoritmo\_genetico: Iterações vs. Tempo (seg.)



Em termos numéricos, pode-se verificar a presença de correlação linear ao utilizar o coeficiente de correlação estatística Pearson, que mede a presença de correlação linear em duas distribuições de valores. Quando o valor se aproxima de 1, tem-se uma forte correlação positiva; quando o valor se aproxima de -1, tem-se uma forte correlação negativa; e quando o valor se aproxima de 0, não há correlação linear.

```
[38]: print(f"Coeficiente de correlação Pearson dos resultados de algoritmo_genetico:␣  
      ↳{np.round(pearsonGA, 3)}")
```

Coeficiente de correlação Pearson dos resultados de algoritmo\_genetico: 0.986

Tem-se constatada, portanto, uma forte correlação linear nos resultados apresentados, embora ligeiramente menor do que a vista no SA.

Vejamos algumas entradas da tabela de resultados, em que algumas soluções produzidas pelo algoritmo podem ser vistas.

```
[39]: tabelaGA
```

```
[39]:
```

	Solução	Função-objetivo	Tempo (seg.)	Iterações
0	[0101, 0001, 1011, 0110, 1010, 0111, 0010, 0100]	0	2.312335	5
1	[0110, 0011, 0001, 1000, 0101, 0010, 0100, 0111]	0	0.082703	0
2	[0011, 0001, 1111, 1000, 0110, 0100, 0010, 0101]	0	1.950228	6
3	[1100, 0101, 1011, 0001, 0110, 1010, 0011, 0111]	0	0.704789	2
4	[0110, 0001, 1010, 0101, 0111, 1100, 0011, 1000]	0	2.307488	7
5	[0000, 1100, 1000, 0110, 0011, 0001, 1110, 0101]	0	3.144580	10
6	[0101, 1100, 0001, 0100, 0111, 0011, 0110, 0010]	0	2.947126	9
7	[1000, 0011, 0111, 0000, 1011, 0110, 0000, 0000]	0	1.346030	4
8	[0101, 0111, 0001, 0100, 0010, 1000, 0110, 0011]	0	0.075392	0
9	[0101, 0111, 0001, 0011, 1000, 0110, 0100, 0010]	0	0.079228	0
10	[0001, 0111, 0000, 1000, 1101, 0010, 0110, 0011]	0	1.657378	5
11	[1111, 1000, 1100, 0011, 0001, 1110, 0010, 0000]	0	1.651467	5
12	[0101, 0011, 1000, 0100, 0111, 0001, 0110, 0010]	0	0.078855	0
13	[0011, 0110, 1000, 0001, 0100, 0111, 0101, 0010]	0	0.077606	0
14	[0101, 0010, 1000, 0011, 0111, 0100, 0001, 1110]	0	3.373618	10
15	[1101, 1111, 0011, 0000, 0110, 1100, 0001, 0101]	0	2.679081	8
16	[0011, 0111, 0010, 0100, 0110, 1100, 0101, 1011]	0	3.451528	11
17	[0101, 1000, 0100, 0001, 0111, 0010, 0110, 1011]	0	0.706135	2
18	[0100, 0110, 1001, 0010, 1101, 0111, 0011, 0001]	0	1.346664	4
19	[1110, 1000, 0100, 1001, 0011, 0110, 0010, 1100]	0	2.302312	7
20	[0011, 0000, 0110, 1111, 1011, 0010, 1000, 0101]	0	1.341734	4
21	[1000, 0011, 0111, 0100, 1010, 0001, 1101, 0110]	0	1.056320	3
22	[0000, 1010, 0001, 1111, 0110, 0000, 0011, 0101]	0	2.295660	7
23	[1000, 0000, 0111, 0011, 0110, 0010, 1111, 1010]	0	6.221395	15
24	[1100, 0010, 0111, 1011, 0000, 0101, 0001, 0110]	0	7.968901	20
25	[0011, 0101, 1101, 0100, 0000, 0111, 1011, 1111]	0	3.267257	10
26	[0110, 0000, 0010, 0111, 1011, 0000, 1110, 0101]	0	3.543428	11
27	[0011, 0001, 0100, 0111, 1100, 0010, 0101, 1000]	0	1.682577	5
28	[0101, 0011, 1000, 0100, 0111, 0001, 1010, 0110]	0	1.107051	3
29	[0011, 1110, 0111, 0010, 0100, 0001, 1000, 0101]	0	1.372144	4
30	[0011, 0001, 1100, 1000, 0101, 0111, 0010, 0110]	0	0.703053	2
31	[0000, 1110, 0101, 0000, 1000, 0100, 0111, 0011]	0	2.589566	8
32	[0010, 0111, 0001, 0011, 0101, 1000, 1110, 0100]	0	1.066177	3
33	[1000, 1111, 0001, 0111, 0101, 0010, 0000, 1101]	0	4.236123	13
34	[0111, 0100, 0001, 0011, 0110, 1001, 1111, 1000]	0	4.476525	14
35	[0010, 0111, 0011, 0110, 1000, 0001, 1101, 0100]	0	0.422873	1
36	[0010, 0110, 1000, 0011, 0001, 0100, 0111, 0101]	0	0.074988	0
37	[1000, 0100, 0000, 1101, 0110, 0001, 0101, 0010]	0	2.297490	7
38	[0010, 0111, 0011, 0110, 1000, 0101, 0001, 0100]	0	0.079139	0
39	[1000, 1010, 0011, 0110, 1011, 0111, 0001, 1101]	0	2.283877	7
40	[0110, 1000, 1011, 0100, 0111, 0101, 1010, 0001]	0	1.325036	4
41	[1110, 0110, 1000, 0000, 0010, 0100, 0111, 1001]	0	2.519844	7
42	[0001, 0101, 0111, 1010, 0100, 0010, 1000, 0011]	0	0.743251	2
43	[1100, 0101, 0111, 1010, 0110, 0010, 1001, 0001]	0	2.021382	6
44	[0101, 0001, 0100, 0110, 1000, 0010, 0111, 0011]	0	0.080754	0
45	[1110, 0111, 0001, 0100, 1000, 0101, 1011, 0010]	0	2.567141	8
46	[0110, 0011, 1100, 0010, 1000, 0101, 0111, 0100]	0	1.367358	4
47	[0000, 1011, 0100, 0110, 0001, 0011, 1010, 1000]	0	2.559863	8
48	[0011, 0101, 0111, 0001, 0100, 0010, 1000, 0110]	0	0.076224	0
49	[0000, 1000, 1110, 0101, 0010, 1001, 0001, 0100]	0	2.633026	8

Por fim, vejamos algumas estatísticas obtidas a partir da tabela de resultados do GA.

```
[40]: print(f"Média e desvio padrão da função-objetivo: {tabelaGA['Função-objetivo'].\n→mean()} ± {tabelaGA['Função-objetivo'].std()} ataques\n"\n      f"Média e desvio padrão do tempo gasto: {np.round(tabelaGA['Tempo (seg.)'].\n→mean(), 3)} ± {np.round(tabelaGA['Tempo (seg.)'].std(), 3)} segundos\n"\n      f"Média e desvio padrão das iterações gastas: {np.\n→round(tabelaGA['Iterações'].mean(), 3)} ± {np.round(tabelaGA['Iterações'].\n→std(), 3)}")
```

Média e desvio padrão da função-objetivo: 0.0 ± 0.0 ataques  
Média e desvio padrão do tempo gasto: 1.925 ± 1.597 segundos  
Média e desvio padrão das iterações gastas: 5.58 ± 4.477

Diferentemente do visto no SA, o número de iterações é muito menor. Contudo, em função da aplicação de operadores inexistentes no SA, tem-se que o tempo de execução médio é aproximadamente de cinco a seis vezes maior, com um desvio correspondente a aproximados 50% deste valor. Esses fatos servem de subsídio para afirmar que este algoritmo, a despeito de definitivamente possuir fatores estocásticos, é mais estável do que o SA em termos de tempo de execução e número de iterações.

## 5 Considerações finais

Com base no exposto acima, tem-se que o problema das oito rainhas foi resolvido de forma ótima por dois algoritmos: *simulated annealing* e *genetic algorithm*. Os dois algoritmos se valem da classe *Tabuleiro*, que concentra a modelagem do problema, para desempenhar suas rotinas. Ambos os algoritmos, com a ajuda das heurísticas implementadas pelos projetistas, foram capazes de encontrar uma solução ótima em todas as suas execuções. Cumpre mencionar que todas as implementações ora expostas foram desenvolvidas almejando total concordância com o estipulado no documento que descreve a atividade.

Durante a exposição dos resultados, foram realizadas análises para verificar a existência de correlação linear entre o número de iterações utilizado e o tempo gasto na execução de ambos os algoritmos. Constatou-se que tanto o recozimento simulado quanto o algoritmo genético implementados possuem forte correlação linear nesse sentido, com especial destaque para o SA.

Em adição, como produtos oriundos da resolução dessas atividades, tem-se algoritmos flexíveis e uma modelagem do problema escalável: a classe *Tabuleiro* foi projetada de forma a suportar diferentes notações e números variados de rainhas e casas no tabuleiro, tal que variações do problema podem ser testadas com alterações mínimas nos algoritmos. Dessa forma, esta base de código pode ser facilmente reciclada para propósitos correlatos.