8 Rainhas

Problema das 8 Rainhas

Figure 1: bg left

O Problema das 8 Rainhas

- **Propósito:** Encontrar uma disposição das 8 rainhas em um tabuleiro de xadrez de forma que nenhuma delas se ataque.
- **Relevância:** Problema clássico de otimização combinatória, com aplicações em jogos e algoritmos genéticos.
- **Abordagem:** Utilização de métodos exatos e heurísticos para encontrar soluções viáveis.

Figure 2: bg right:30%

Descrição do Problema

- **Tabuleiro de Xadrez:** 8x8
- **Rainhas:** 8

- **Objetivo:** Dispor as 8 rainhas de forma que nenhuma delas se ataque.
- **Restrições:** Nenhuma rainha pode atacar outra na mesma linha, coluna ou diagonal.
- **Generalização:** O problema pode ser generalizado para tabuleiros de tamanho $n \times n$ e n rainhas.

Figure 3: bg left:30%

Implementação em Python

- Funções preliminares
  - **Estrutura de dados**: O que vamos usar para representar o tabuleiro?
  - **Função de vizualização**: Como vamos mostrar o tabuleiro?

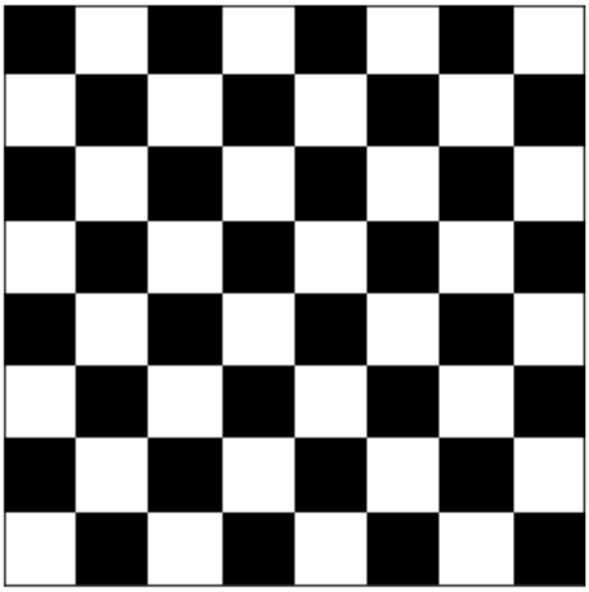

Figure 4: bg fit

– **Função de avaliação**: Como vamos determinar se uma solução é válida?

Estrutura de Dados

Estrutura elementar

A forma mais direta de se representar um tabuleiro de xadrez é através de uma matriz 8x8. Cada posição da matriz pode ser ocupada por uma rainha ou não.

```
import numpy as np
tabuleiro = np.zeros((8, 8), dtype=bool)
```

Um pouco de combinatória

- Chamaremos de **solução** qualquer disposição das 8 rainhas no tabuleiro.
- Chamaremos de **solução válida** aquela em que nenhuma rainha se ataca.

Quantas **soluções** existem para o problema das 8 rainhas?

Em um tabuleiro de xadrez 8x8, existem $\binom{64}{8}$ possíveis disposições de 8 rainhas.

4.426.165.368

Usando um pouco a cabeça

- Muitas destas soluções são obviamente inválidas.
  - Aquelas que tem duas rainhas na mesma linha ou coluna.
- Sabemos que exite exatamente uma rainha em cada linha e coluna. Podemos usar o princípio da multiplicação para calcular o número de soluções sem ataque em linhas ou colunas.
  - d_1: Qual linha ocupar na coluna 1? *–8 possibiliades–*
  - d_2: Qual linha ocupar na coluna 2? *–7 possibiliades–*
  - ...
  - d_8: Qual linha ocupar na coluna 8? *–1 possibiliade–*
- Pelo princípio da multiplicação, o número total de soluções é $8! = 40.320$.

Repensando a estrutura de dados

- Parece desperdício usar uma matriz 8x8 para este problema.
- Uma estrutura capaz de armazenar 4.426.165.368 quando só precisamos de 40.320, ainda menos.
- Nós só precisamos indicar a linha em que cada rainha está para cada coluna.
- Uma permutação de 8 elementos basta, mas não há esse tipo de estrutura em Python, então usaremos um array de inteiros.

```
import numpy as np
solucao = np.array([0, 1, 2, 3, 4, 5, 6, 7], dtype=int)
```

- Faremos que se comportem como permutações via lógica de programação.

Função de Vizualização

Básico

```
def printar(solucao):
    n = len(solucao)
    for i in range(n):
        for j in range(n):
            if solucao[j] == i:
                print('\u2655', end='')
            else:
                print('\u2B1C', end='')
```

```
        print()
    print()
```

- **Os caracteres `'\u2655'` e `'\u2B1C'` são respectivamente o símbolo de uma rainha e um quadrado vazio.**

Avançado

- Vamos usar a biblioteca `matplotlib`

```
import matplotlib.pyplot as plt
solucao = np.array([0, 1, 2, 3, 4, 5, 6, 7], dtype=int)
def plotar(solucao):
    n = len(solucao)
    dx, dy = 0.015, 0.05
    x = np.arange(-4.0, 4.0, dx)
    y = np.arange(-4.0, 4.0, dy)
    X, Y = np.meshgrid(x,y)
    min_max = np.min(x), np.max(x), np.min(y), np.max(y)
    res = np.add.outer(range(n), range(n))%2
    plt.imshow(res, cmap="binary_r")
    plt.xticks([])
    plt.yticks([])
    for i in range(n):
        plt.text(i, solucao[i], '\u2655', color="orange", fontsize=20, ha="center", va="cent
    plt.show()
```

Função de Avaliação

- Há três tipos de problemas na computação:
  - **Satisfação:** Existe uma solução?
  - **Localização:** Onde está uma solução?
  - **Otimização:** Qual é a melhor solução?
- O problema das 8 rainhas é um problema de localização.
- Mas podemos transformá-lo em um problema de otimização.

Função de Avaliação

- A função de avaliação é uma função que atribui um valor numérico a uma solução.
- Vamos contar o número de ataques entre as rainhas. Considerando que não há ataques entre rainhas na mesma linha ou coluna, só precisamos verificar as diagonais.

```
def avaliar(solucao):
    ataques = 0
    n = len(solucao)
    for i in range(n):
        for j in range(i+1, n):
            if solucao[i] > 0 and solucao[j] > 0:
                if abs(solucao[i] - solucao[j]) == j - i:
                    ataques += 1
    return ataques
```

Solução por Força Bruta

- Vamos gerar todas as soluções possíveis e avaliá-las.
- Se encontrarmos uma solução sem ataques, podemos parar a busca.
- Vamos usar a biblioteca `itertools` para gerar todas as permutações possíveis.

```
import itertools
def forca_bruta(n=8):
    melhor_solucao = np.array(range(n), dtype=int)
    menor_ataques = avaliar(melhor_solucao)
    for solucao in itertools.permutations(range(n)):
        ataques = avaliar(solucao)
        if ataques < menor_ataques:
            menor_ataques = ataques
            melhor_solucao = solucao
            if ataques == 0:
                break
    return melhor_solucao
```

Gerador Apenas Permutações Válidas

- Vamos usar um algoritmo recursivo para gerar todas as permutações possíveis de um conjunto.

```
def forca_bruta_otimizada(n=8):
    solucao = np.full(n, -1, dtype=int)
    usados = np.full(n, False, dtype=bool)
    for i in range(n):
        if coloca(i,0, solucao, usados):
            break
    return solucao
```

- A função `ha_ataque` verifica se uma rainha na coluna `coluna` ataca alguma outra rainha.
- É uma versão modificada da função `avaliar` que verifica se uma solução é válida.

- Esta função leva em conta apenas as rainhas que já foram colocadas em relação com a última.

```
def ha_ataque(coluna, solucao):
    for i in range(coluna):
        if abs(solucao[i] - solucao[coluna]) in (0, coluna - i):
            return True
    return False
```

Abordagem Aleatória

- Vamos gerar soluções aleatórias e avaliá-las.Se encontrarmos uma solução sem ataques, podemos parar a busca.
- Vamos usar um limite de iterações para evitar que o algoritmo fique preso em um loop infinito.

```
def aleatoria(n=8, max_iter=1000):
    solucao = np.array(range(n), dtype=int)
    melhor_solucao = np.array(range(n), dtype=int)
    menor_ataques = avaliar(melhor_solucao)
    for _ in range(max_iter):
        np.random.shuffle(solucao)
        ataques = avaliar(solucao)
        if ataques < menor_ataques:
            menor_ataques = ataques
            melhor_solucao = solucao
            if ataques == 0:
                break
    return melhor_solucao
```

Vantagens e Desvantagens

- **Vantagens:**
  - Fácil de implementar.
  - Rápido, o tempo de execução não aumentata tando com o incremento de `n`.
  - O tempo de execução é controlável.
  - Pode ser usado como ponto de partida para algoritmos mais sofisticados.
- **Desvantagens:**
  - Não garante a solução ótima.

Abordagem Gulosa

- Vamos usar uma abordagem gulosa para resolver o problema.

- **A ideia é colocar uma rainha em cada coluna, escolhendo a linha que minimiza o número de ataques.**

```
def gulosa(n=8):
    solucao = np.full(n,-1, dtype=int)
    usadas = np.full(n, False, dtype=bool)
    ataques = 0
    for coluna in range(n):
        melhor_linha = -1
        menor_ataques = n
        for linha in range(n):
            if not usadas[linha]:
                solucao[coluna] = linha
                av = avaliar(solucao)
                if av < menor_ataques:
                    menor_ataques = av
                    melhor_linha = linha
                    if menor_ataques == ataques:
                        break
        solucao[coluna] = melhor_linha
        usadas[melhor_linha] = True
        ataques = menor_ataques
    return solucao, ataques
```

Vantagens e Desvantagens

- **Vantagens:**
  - Rápido, o tempo de execução não aumentata tando com o incremento de `n`.
  - O tempo de execução é controlável.
  - Pode ser usado como ponto de partida para algoritmos mais sofisticados.
- **Desvantagens:**
  - Não garante a solução ótima.
  - Gera sempre a mesma solução para um mesmo tabuleiro (determinístico).
- Como podemos melhorar?

Gulosa Aleatorizada

- Ao invés de seguir a ordem das colunas, vamos escolher aleatoriamente a ordem das colunas.

- Também vamos escolher aleatoriamente a ordem das linhas para cada coluna.

```
def gulosa_randomizada(n=8):
    solucao = np.full(n,-1, dtype=int)
    usadas = np.full(n, False, dtype=bool)
    limite_inferior = 0
    colunas = np.arange(n)
    linhas = np.arange(n)
    np.random.shuffle(colunas)
    for coluna in colunas:
        melhor_linha = -1
        menor_ataques = n
        np.random.shuffle(linhas)
        for linha in linhas:
            if not usadas[linha]:
                solucao[coluna] = linha
                av = avaliar(solucao)
                if av < menor_ataques:
                    menor_ataques = av
                    melhor_linha = linha
                    if menor_ataques == limite_inferior:
                        break
        solucao[coluna] = melhor_linha
        usadas[melhor_linha] = True
        limite_inferior = menor_ataques
    return solucao, limite_inferior
```

- Como cada execução do algoritmo gera uma solução diferente, podemos executá-lo várias vezes e escolher a melhor solução.

```
def gulosa_randomizada_repetida(n=8, max_iter=100):
    melhor_solucao = np.full(n,-1, dtype=int)
    menor_ataques = n
    for _ in range(max_iter):
        solucao, ataques = gulosa_randomizada(n)
        if ataques < menor_ataques:
            menor_ataques = ataques
            melhor_solucao = solucao
            if ataques == 0:
                break
    return melhor_solucao, menor_ataques
```

Vantagens e Desvantagens

- **Vantagens:**
  - Rápido, o tempo de execução não aumentata tando com o incremento de `n`.
  - O tempo de execução é controlável.
  - Pode ser usado como ponto de partida para algoritmos mais sofisticados.
  - Gera soluções diferentes para um mesmo tabuleiro (estocástico).
- **Desvantagens:**
  - Não garante a solução ótima.

Melhorando Uma Solução Existente

- Uma outra abordagem é melhorar uma solução existente.
- Vamos usar a abordagem gulosa para gerar uma solução inicial e depois melhorá-la.
- Vamos tentar melhorar a solução trocando as posições de duas rainhas.

```
def melhorar(solucao, ataques):
    n = len(solucao)
    for i in range(n):
        for j in range(i+1, n):
            solucao[i], solucao[j] = solucao[j], solucao[i]
            novos_ataques = avaliar(solucao)
            if novos_ataques < ataques:
                return True, novos_ataques
            solucao[i], solucao[j] = solucao[j], solucao[i]
    return False, ataques
```

Ladeira Abaixo ou Colina acima

- Por quê se contentar com uma melhoria?
- Vamos continuar trocando as posições das rainhas até que não seja mais possível melhorar a solução.
- Este tipo de algoritmo é chamado de **Método de descida** ou **Hill Climb**.

```
def hill_climb(solucao, ataques):
    while True:
        melhorou, ataques = melhorar(solucao, ataques)
        if not melhorou:
            break
    return solucao, ataques
```

Vantagens e Desvantagens

- **Vantagens:**
  - Consegue melhorar soluções existentes e obter soluções de boa qualidade mesmo para valores maiores de `n`.
  - Podemos repetir o algoritmo várias vezes, usando soluções inicias diferentes, e escolher a melhor solução.
- **Desvantagens:**
  - Não garante a solução ótima.

Obrigado!