Problema:
https://pt.wikipedia.org/wiki/Problema_das_oito_damas

Problema das N-Rainhas

Gustavo Silva Resende[1], Luis Gustavo de Souza[2], Omar Condori Lopez[3]

Universidade Tecnológica Federal do Paraná - UTFPR. Câmpus: Cornélio Procópio.

1 Estado Inicial

Para o início do problema é necessário a criação do estado inicial. Esta simples função foi elaborada através de uma sequência aleatória que gera N números de 0 até N. A variável N trata-se de um número inteiro que representa a dimensão da problemática, ou seja, se N=8, então a situação a se resolver será de 8-Rainhas.

```python
def estado_inicial(n = 8):
    ''' Seta o estado inicial de maneira randômica

        Arguments: n (int) -> número de rainhas
        Return: Matriz com as posições das rainhas como 1, e sem rainha como 0
    '''

    matriz_inicial = np.zeros((n,n))
    random_indices = np.random.randint(n, size=n)

    for i, r in enumerate(random_indices):
        matriz_inicial[r, i] = 1

    return matriz_inicial.astype('int')
```

Fig. 1: Função responsável por gerar o estado inicial

Com a função representada pela Figura 1, nota-se que é criado uma matriz de zeros $N \times N$, e para cada coluna será colocado uma rainha (representado pelo valor 1) através da sequência aleatória gerada. Após isso, a função retorna a matriz do estado inicial das rainhas.

2 Custo/Conflitos do Estado Atual

Esta função é responsável por verificar o número de pares de rainhas em conflito no estado atual. O número de conflitos é calculado de forma independente para as linhas e diagonais, sendo somados no fim da execução para obter-se o número de conflitos total do estado.

```python
def custo_quadrado(estado, n):
    ''' Calcula o custo de matchs entre rainhas de uma determinada matriz

        Arguments:  estado (matriz int) -> Matriz com as posições das rainhas
                    n (int) -> número de rainhas
        Return: Float que representa o custo dessa matriz
    '''


    custo_diagonal = 0
    rainhas = np.argwhere(estado == 1)

    soma_hor = np.sum(estado, axis=1)

    for i, s in enumerate(soma_hor):
        soma_hor[i] = np.sum(np.arange(s))
    soma_hor = np.sum(soma_hor)

    for i in range(n):
        for j in range(i, n):
            x1, y1 = rainhas[i][0], rainhas[j][0]
            x2, y2 = rainhas[i][1], rainhas[j][1]
            if ((x1 + x2) == (y1 + y2) or (x1 - x2) == (y1 - y2)) and i!=j:
                custo_diagonal+=1

    return custo_diagonal+soma_hor
```

Fig. 2: Função responsável por calcular os conflitos do estado atual.

3 Custo da Matriz Total

O objetivo desta função é calcular o número de conflitos dos estados vizinhos ao atual, para isto a posição das rainhas é trocada, uma por vez. Após a realização de uma troca, o número de conflitos do novo estado e calculado e armazenado em uma matriz. Tal valor é gravado na mesma posição que a rainha que teve sua posição alterada. Com o número de conflitos deste estado armazenado, a posição original das rainhas é restaurada. Este processo se repete até que todas as rainhas tenham sido movidas para todas as posições de sua coluna e os seus respectivos conflitos tenham sido calculados e armazenados na matriz.

```python
def matriz_custo_total(estado, n):
    ''' Calcula uma matriz custo de todas as possíveis movimentações

        Arguments:  estado (matriz int) -> Matriz com as posições das rainhas
                    n (int) -> número de rainhas
        Return: Matriz custo de todas as possíveis movimentações
    '''

    custo_total = np.zeros((n, n))
    estado_certo = estado.copy()

    for i in range(n):
        for j in range(n):
            estado[:, i] = 0
            estado[j][i] = 1
            custo_total[j][i] = custo_quadrado(estado, n)
        estado = estado_certo.copy()

    return custo_total
```

Fig. 3: Função responsável por calcular os conflitos dos estados vizinhos.

4 Hill Step - Próximo Passo

A função presente tem o objetivo de mover uma rainha para posição de menor custo de acordo com a matriz de custo total. A função, primeiramente, resgatará as posições das rainhas na matriz de estado, e após isso, modificará a matriz custo nas posições das rainhas, colocando um custo alto (como no caso: n*n), para que o argumento mínimo da matriz custo não esteja na mesma posição de uma rainha, obrigando a movimentação da mesma. Após isso, o estado atual será atualizado.

```python
def hill_step(estado_atual, controle, n):
    ''' Realiza uma movimentação dado a melhor opção da matriz custo

        Arguments:  estado_atual (matriz int) -> Matriz com as posições das rainhas
                    controle (int) -> Variável que controla se o algoritmo deve ou não tomar uma decisão randômica
                    n (int) -> número de rainhas
        Return: Estado atual, custo atual e a variável controle
    '''

    matriz_custo = matriz_custo_total(estado_atual, n)
    rainhas = np.argwhere(estado_atual == 1)

    for r in rainhas:
        matriz_custo[r[0]][r[1]] = n*n

    valor_minimo = np.min(matriz_custo)
    indices_minimo = np.argwhere(matriz_custo == valor_minimo)
    choose_one = np.random.randint(len(indices_minimo))
    x, y = indices_minimo[choose_one]

    for r in rainhas:
        estado_atual[:, y] = 0
        estado_atual[x][y] = 1
        break

    custo_atual = custo_quadrado(estado_atual, n)

    if controle >= n:
        x, y = np.random.randint(0, n, size=2)
        estado_atual[:, y] = 0
        estado_atual[x][y] = 1
        controle = 0

    controle += 1

    return estado_atual, custo_atual, controle
```

Fig. 4: Função responsável por fazer um movimento de uma rainha.

Se não for encontrado a solução em *N* passos, o algoritmo realiza uma movimentação aleatória, a fim de escapar dos mínimos locais. Esse movimento é controlado por uma variável que é incrementada a cada iteração.

5 Hill Climb

Esta função construirá a heurística do Hill Climb, que consiste em movimentar-se a rainha para o menor local possível, fazendo isso até encontrar o custo zero, que representa a solução do problema.

```python
def hill_climb(n):
    ''' Realiza uma sequência de passos até concluir o objetivo de posicionar as rainhas

        Arguments: n (int) -> número de rainhas
        Return: Estado Final com as rainhas posicionadas corretamente
    '''

    estado_atual = estado_inicial(n)
    custo_atual = custo_quadrado(estado_atual, n)
    controle = 0

    while custo_atual != 0:
        estado_atual, custo_atual, controle = hill_step(estado_atual, controle, n)
        print("Estado:\n{}\n\nCusto:\n{}\n\n".format(estado_atual, custo_atual))

    return estado_atual
```

Fig. 5: Função responsável por gerar uma sequência de movimentos das rainhas, afim de encontrar a solução.

6 Simulação

Para questão de exemplificação do funcionamento do algoritmo, foi criado a seção presente para simular a execução do algoritmo.

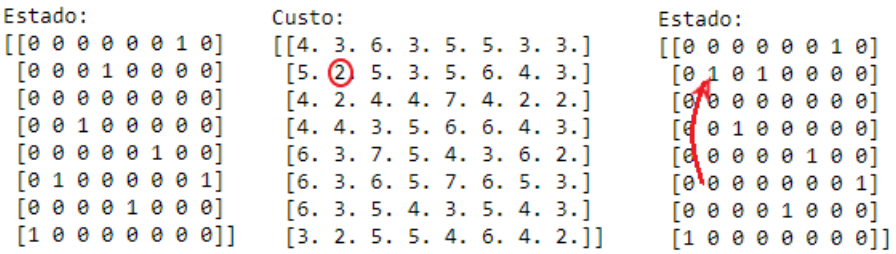

Fig. 6: Simulação do movimento de uma rainha pelo menor caminho.

Sendo que, esta ação da Figura 6 irá se repetir sequencialmente até atingir a solução. Porém, como já foi dito, em determinados eventos, exige-se a necessidade de gerar um movimento aleatório para prevenir-se dos mínimos locais. Após a ação aleatória o algoritmo continua com sua heurística inicial de escolher o melhor caminho. Como demonstra a Figura 7.

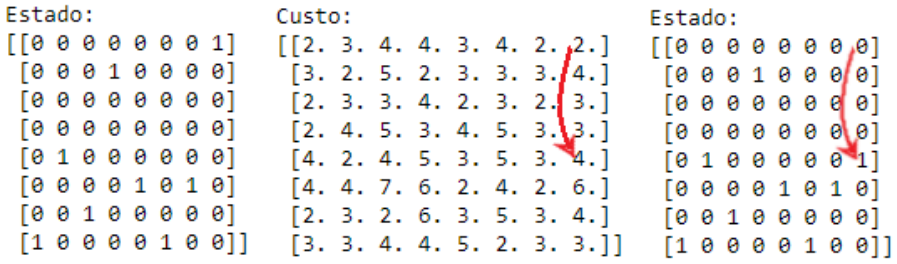

Fig. 7: Simulação do movimento aleatório de uma rainha

7 Performance

Em muitos casos, o algoritmo não cai em um local mínimo, resolvendo um problema de N-rainhas facilmente. Porém, em determinados eventos é inevitável o mesmo, fazendo com que o desempenho caia drasticamente. Para estabelecer uma métrica de qualidade do algoritmo, construiu-se a seguinte tabela executando o algoritmo 50 vezes.

Tempo Médio de Execução	0.59s
Tempo Médio de Execução sem Movimentos Aleatórios	0.11s
Tempo Médio Acrescentado por Movimento Aleatório	0.12s
Média de Iterações por Execução	32.3
Média de Iterações sem Movimentos Aleatórios	5.25
Média de Iterações Acrescentada por Movimentos Aleatórios	6.92
Média de Movimentos Aleatórios Gerados	4.08

Tab. 1: Tabela construída sobre 50 execuções do algoritmo.