

Busca Tabu para N-rainhas

Gustavo Silveira Dias
Bacharelado em Engenharia de Computação
Instituto Federal de Minas Gerais
Inteligência Artificial
Prof. Dr. Cíniro Nametala

SUMÁRIO

I	Introdução	1
II	Definição do problema	1
III	Metodologia	1
III-A	Principais modificações	1
III-B	Parametrização	2
IV	Resultados	2
V	Conclusões	2
	Referências	2

LISTA DE FIGURAS

1	Avaliar colisão.	1
2	Análise de colisões	1
3	Gerar solução inicial	2

Busca Tabu para N-rainhas

Resumo—Neste relatório técnico são discutidos detalhes de implementação, testes e resultados obtidos com o algoritmo Busca Tabu quando aplicado ao problema das N-rainhas. Buscando-se balancear parâmetros foram realizadas avaliações observando a quantidade de conflitos da solução. Após definição de parâmetros, o algoritmo balanceado foi executado e os resultados mostraram que o número de colisões sempre está próximo de 0. Nesse sentido, a principal conclusão deste experimento é que o algoritmo da Busca Tabu tem uma grande capacidade de encontrar soluções boas, mas é necessário que a quantidade de rainhas despostas não seja tão elevada.

I. INTRODUÇÃO

O problema das N-rainhas tem como objetivo posicionar N rainhas em um tabuleiro de xadrez $N \times N$ de forma que nenhuma rainha ataque a outra. A busca tabu é utilizada para explorar diferentes disposições das rainhas de maneira eficiente, evitando soluções inviáveis e promovendo a exploração de soluções de qualidade. Além disso, o algoritmo de busca mantém uma lista que armazena movimentos já testados que não devem ser repetidos temporariamente.

Inicialmente o algoritmo considera uma solução inicial qualquer como a solução atual logo após, é gerado soluções vizinhas que é aquela que possui uma leve modificação em relação a solução atual, ou seja, após a troca ser realizada, deverá ser armazenado o movimento realizado. Esse movimento é chamado de movimento tabu, pois não poderá ser repetido por um período. Depois, de gerar vizinhos deve-se encontrar um que seja melhor que a solução atual. Quando isso acontecer, esse melhor vizinho deve substituir a solução atual como uma nova melhor solução, além disso a cada tentativa de gerar uma melhor solução deve sempre ir atualizando a lista tabu com o movimento que foi realizado.

Toda a implementação foi realizada em linguagem de programação Python (Python 3.11.1) e o *hardware* utilizado foi um notebook Acer com processador i5 de 10 geração e 8GB de memória RAM.

II. DEFINIÇÃO DO PROBLEMA

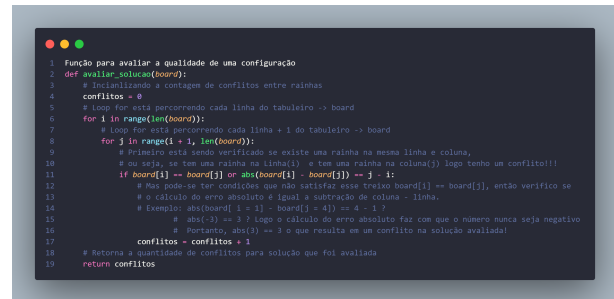
Dado um tabuleiro de xadrez $N \times N$, o objetivo é posicionar N rainhas no tabuleiro de forma que nenhuma rainha possa atacar a outra. Isso significa que duas rainhas não podem compartilhar a mesma linha, coluna ou diagonal no tabuleiro. O problema consiste em encontrar uma disposição das N rainhas que satisfaça essa condição ou que se aproxime dessa melhor solução.

III. METODOLOGIA

O algoritmo de Busca Tabu implementado para avaliação neste relatório seguiu os mesmos passos propostos e já comentados na seção de introdução, as principais modificações e a parametrização são apresentadas nas subseções a seguir.

A. Principais modificações

Como pode ser visto na Figura 1 algoritmo foi modificado para incorporar uma função encarregada de avaliar as colisões entre as soluções geradas, sem a necessidade de subdividir o tabuleiro em diagonais positivas e negativas. No entanto, após conduzir uma série de testes, foi identificado desafios significativos na detecção de soluções que possuem valores de entrada muito elevado.



```

1 Função para avaliar a qualidade de uma configuração
2 def avaliar_solucao(board):
3     # Calculando a contagem de conflitos entre rainhas
4     conflitos = 0
5     # Loop for está percorrendo cada linha do tabuleiro -> board
6     for i in range(len(board)):
7         # Loop for está percorrendo cada coluna + 1 do tabuleiro -> board
8         for j in range(i + 1, len(board)):
9             # Primeiro está sendo verificado se existe uma rainha na mesma linha e coluna,
10             # ou seja, se tem uma rainha na linha(i) e tem uma rainha na coluna(j) logo tem um conflito!!
11             if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
12                 # Mas poderia ter condições que não satisfizesse esse trecho board[i] == board[j], então verifico se
13                 # o cálculo do erro absoluto é igual a subtração de coluna - linha.
14                 # Exemplo: abs(board[1] - board[4]) == 4 - 1
15                 # abs(-3) == 3 logo o cálculo do erro absoluto faz com que o número nunca seja negativo
16                 # Portanto, abs(i) == 3 o que resulta em um conflito na solução avaliada!
17                 conflitos = conflitos + 1
18     # Retorna a quantidade de conflitos para solução que foi avaliada
19     return conflitos

```

Figura 1: Avaliar colisão.

Portanto, como pode ser visto na Figura 2 a função foi alterada para atuar de maneira otimizada. Para isso, a função rastreia as diagonais positivas e negativas, analisando cada coluna do tabuleiro e determinando se múltiplas rainhas ocupam a mesma diagonal, incrementando o contador conforme é necessário. O resultado final de conflitos resultante é usada para orientar a busca por uma solução livre de conflitos ou que tenha uma quantidade de conflitos baixa.



```

1 def avaliar_solucao(tabuleiro):
2     # Obter o tamanho do tabuleiro (número de rainhas e colunas).
3     n = len(tabuleiro)
4
5     # Inicializar listas para rastrear as diagonais positivas e negativas.
6     diagonal_positiva = [0] * (2 * n - 1)
7     diagonal_negativa = [0] * (2 * n - 1)
8
9     # Inicializar variável para contar conflitos.
10    conflitos = 0
11
12    # Iterar pelas colunas do tabuleiro.
13    for i in range(n):
14        # Calcular índices das diagonais positivas e negativas para a rainha na coluna 'i'.
15        indice_positivo = tabuleiro[i] + i
16        indice_negativo = tabuleiro[i] - i + n - 1
17
18        # Verificar se os índices estão dentro dos limites das listas de diagonais.
19        if 0 <= indice_positivo < 2 * n - 1:
20            diagonal_positiva[indice_positivo] += 1
21
22        if 0 <= indice_negativo < 2 * n - 1:
23            diagonal_negativa[indice_negativo] += 1
24
25    # Iterar pelas diagonais para contar conflitos.
26    for i in range(2 * n - 1):
27        if diagonal_positiva[i] > 1:
28            # Se mais de uma rainha estiver em uma diagonal positiva, incrementar conflitos.
29            conflitos += diagonal_positiva[i] - 1
30
31        if diagonal_negativa[i] > 1:
32            # Se mais de uma rainha estiver em uma diagonal negativa, incrementar conflitos.
33            conflitos += diagonal_negativa[i] - 1
34
35    # Retornar o número total de conflitos encontrados no tabuleiro.
36    return conflitos
37

```

Figura 2: Análise de colisões

Nesse contexto, é relevante destacar que a implementação do algoritmo incorpora verificações de conflitos entre as rainhas. Contudo, como evidenciado na Figura 3, foi adotada uma abordagem na qual uma função é empregada para gerar números aleatórios utilizando a biblioteca Random do Python 3, juntamente com a função Sample. Assim, a solução inicial gerada por esse método assegura a ausência de quaisquer conflitos nas colunas do tabuleiro.

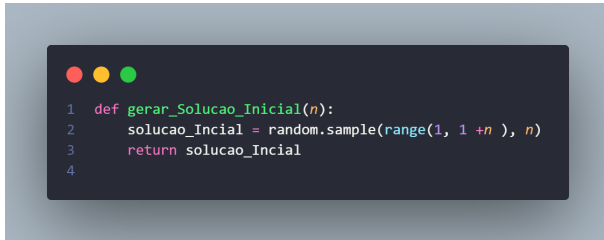


Figura 3: Gerar solução inicial

A implementação incluiu um passo opcional, que consiste na incorporação de um critério de aspiração. Esse critério foi aplicado para verificar se a quantidade de conflitos na solução atual era, de fato, menor que a quantidade de conflitos que na solução que está sendo avaliada.

B. Parametrização

Foram realizados testes variando-se, vez a vez, cada um dos parâmetros para se observar comportamentos como o número de conflitos da solução e também a velocidade *segundos* que o algoritmo encontra a solução. Após realizar 15 testes, a parametrização utilizada foi a seguinte:

- Movimento tabu: 5 *Iteração*
- Máximo de iterações: 1000 *Iteração*
- Taxa de tempo para encontrar a solução: 0.10 *segundos*

IV. RESULTADOS

Como pode ser observado na Tabela 1, os resultados demonstram que o algoritmo implementado consistentemente alcança valores substancialmente baixo para algumas das entradas apresentas.

Ao efetuar uma breve análise comparativa entre as entradas com N igual a 6 e N igual a 8, foi notado que o algoritmo raramente converte para uma solução sem conflitos quando N é igual a 6. Por outro lado, é consistentemente bem-sucedido na obtenção de uma solução ótima, sem qualquer conflito, para entradas com N igual a 8.

Portanto, é evidente que ainda podem existir técnicas de codificação que possibilitem uma otimização mais eficaz na verificação de soluções. Este aspecto se torna particularmente notável quando se considera que, para entradas com N igual a 6, a quantidade de conflitos é notavelmente baixa, e, portanto, seria de esperar uma resolução livre de conflitos com grande facilidade.

Tabela I: Resultados da Busca

Entrada	Conflitos
6	1
8	0
28	0
30	0
32	0
34	1
40	1
50	1
70	1
100	1

V. CONCLUSÕES

Neste relatório, o algoritmo de Busca Tabu aplicado ao problema das N -rainhas é analisado detalhadamente. O algoritmo foi implementado e avaliado quanto a sua capacidade de encontrar soluções sem conflitos para diferentes valores de N . Os resultados mostra padrões notáveis em relação a eficácia do algoritmo, com desempenho variando significativamente de acordo com o tamanho do tabuleiro. Toda a análise também indicou a possibilidade de aprimoramentos da codificação e na estratégia de busca para soluções mais rápidas e eficientes, especialmente para tabuleiros pequenos. Em resumo, este estudo proporcionou uma compreensão aprofundada da aplicação da Busca Tabu ao problema e apontou direções para uma futura melhoria do código.

REFERÊNCIAS

- [1] SRINIVAS, N., DEB, K. "Multiobjective optimization using nondominated sorting in genetic algorithms". *Evolutionary Computation*. V. 2, n. 3, p. 221-248, 1994.