Implementação e Otimização do Algoritmo das N-Rainhas com Busca Tabu

Hugo César Leal
Bacharelado em Engenharia de Computação
Instituto Federal de Minas Gerais
Inteligência Artificial
Prof. Dr. Ciniro Nametala

SUMÁRIO

LISTA DE FIGURAS

Implementação e Otimização do Algoritmo das N-Rainhas com Busca Tabu

***Resumo*—Este relatório apresenta a evolução de duas implementações do algoritmo de Busca Tabu aplicadas ao problema das N-Rainhas. A versão final utiliza uma abordagem que foca na troca de rainhas em conflito, o que resultou em uma melhora significativa de desempenho em comparação à implementação inicial com listas de controle. São discutidos detalhes das funções de avaliação de conflitos e da geração de vizinhos, com uma análise comparativa entre as duas abordagens.**

I. Introdução

A Busca Tabu é uma metaheurística para problemas de otimização que previne a repetição de soluções e evita mínimos locais ao armazenar soluções já visitadas em uma lista tabu. No problema das N-Rainhas, busca-se dispor N rainhas em um tabuleiro $N \times N$ sem que compartilhem a mesma linha, coluna ou diagonal.

Inicialmente, foi sugerido pelo professor o uso de uma matriz para controlar as trocas, onde o triângulo superior indicaria até qual iteração uma troca ficaria travada e o triângulo inferior contaria a frequência das trocas. No entanto, para reduzir a complexidade, optei por implementar uma primeira versão com duas listas separadas para monitorar as trocas e as iterações de travamento, e uma versão final que foca nas trocas entre rainhas em conflito. Todos os algoritmos foram implementados em Python 3.9.

II. Definição do Problema

O problema das N-Rainhas consiste em minimizar os conflitos entre rainhas, conforme a Equação 1:

$$f(x) = \sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} Conf(x_i, x_j), \tag{1}$$

onde $Conf(x_i, x_j) = 1$ se x_i e x_j estão em conflito e 0 caso contrário.

III. Metodologia

Esta seção descreve o processo de implementação e as melhorias introduzidas nas duas versões do algoritmo.

A. Primeira Implementação: Listas de Controle para Trocas e Travamento

A primeira implementação utilizou duas listas para gerenciar trocas e travamentos. Uma lista registrava as trocas realizadas e suas respectivas contagens, enquanto a outra armazenava o número de iterações de travamento associadas a cada troca. Esse método reduziu consideravelmente a complexidade em comparação à implementação sugerida pelo professor, que utilizava uma matriz para armazenar essas informações. No entanto, essa abordagem ainda exigia um monitoramento constante das trocas e travamentos.

B. Segunda Implementação: Troca Focada em Rainhas em Conflito

Na segunda e mais eficiente implementação, simplifiquei o algoritmo ao eliminar as listas de travamento e focar exclusivamente nas rainhas em conflito. Abaixo, descrevo os principais detalhes dessa abordagem.

- Função de Avaliação (funcaoObjetivo): A função identifica conflitos ao verificar se duas rainhas compartilham a mesma diagonal, armazenando essas informações em conjuntos para facilitar a checagem. Além de retornar o número de conflitos, a função gera uma lista das rainhas em conflito, utilizada na geração de vizinhos.
- Geração de Vizinhos (gerarVizinho): Nesta versão, a função gerarVizinho realiza trocas apenas entre posições de rainhas em conflito. Uma posição é escolhida aleatoriamente entre as rainhas em conflito e outra posição aleatória é selecionada para a troca, o que reduz o número de trocas e mantém o foco nas rainhas que causam problemas.
- Loop de Busca Tabu (buscaTabu): A busca itera até que todos os conflitos sejam resolvidos ou o número máximo de iterações seja atingido. Em cada iteração, um vizinho é gerado e, se este reduz o número de conflitos, substitui a solução atual. Também é possível que a busca seja executada quantas vezes forem necessárias para encontrar a fitness 0, através da variável rodarAteEncontrarZero.

C. Critério de Aspiração

O critério de aspiração no algoritmo de Busca Tabu permite que movimentos bloqueados pela lista tabu sejam reavaliados e possivelmente aceitos caso atendam a condições específicas que os tornem vantajosos. Um exemplo disso ocorre quando todas as trocas possíveis estão bloqueadas; nesse caso, o critério de aspiração pode liberar a troca da posição menos frequente até o momento. Esse critério é particularmente útil em problemas onde a exploração da vizinhança frequentemente leva a mínimos locais, ajudando o algoritmo a escapar dessas regiões e a prosseguir em busca de soluções mais promissoras.

1) Aplicação do Critério de Aspiração no Problema das N-Rainhas: No entanto, no contexto do problema das N-Rainhas, o uso do critério de aspiração não foi necessário. Isso ocorre porque, na implementação otimizada, a geração de vizinhos é restrita apenas às rainhas em conflito, o que minimiza o risco de revisitar configurações indesejáveis ou pouco eficientes. Dessa forma, o algoritmo evita mínimos locais sem a necessidade de desbloquear soluções tabu, focando diretamente na resolução dos conflitos presentes e garantindo uma convergência mais rápida e eficiente. Portanto, a estrutura do código e a estratégia de foco nas rainhas conflitantes

tornaram dispensável o critério de aspiração, simplificando a implementação e mantendo o desempenho elevado.

D. Parametrização

Para o funcionamento do algoritmo de Busca Tabu no contexto das N-Rainhas, são usados os seguintes parâmetros:

- Número de Iterações Máximas: Define o número máximo de vezes que o algoritmo tentará encontrar uma solução sem conflitos, prevenindo um loop infinito em casos onde a solução é difícil de otimizar.
- Tamanho do Tabuleiro: Determina o valor de N, ou seja, a quantidade de rainhas e o tamanho do tabuleiro $N \times N$, o que impacta a complexidade do problema e o tempo de execução.
- Execução até Solução sem Conflitos: Este parâmetro controla se o algoritmo deve continuar a busca até encontrar uma solução perfeita ou se pode encerrar ao atingir o limite de iterações. Isso permite definir o nível de exaustividade da busca.

IV. Resultados

Para $N = 1000$, a segunda implementação demonstrou uma convergência significativamente mais rápida em relação à primeira, conforme ilustrado na Figura 1 (Gráfico de Convergência da Primeira Implementação) e na Figura 2 (Gráfico de Convergência da Segunda Implementação). Os experimentos foram executados em um processador Ryzen 5 5500U. Nesse ambiente, a primeira implementação apresentou um tempo médio de 450 segundos, enquanto a segunda implementação levou apenas 1,5 segundo para resolver o problema das 1000 rainhas.

Vale destacar que os testes foram realizados até que o algoritmo atingisse uma solução com fitness igual a 0. Nesse processo, a primeira implementação exigiu aproximadamente 160.000 iterações para convergir, ao passo que a segunda, ao focar apenas nas rainhas em conflito, alcançou o mesmo resultado em cerca de 7.000 iterações. A Tabela 1 resume a comparação de desempenho entre as duas abordagens.

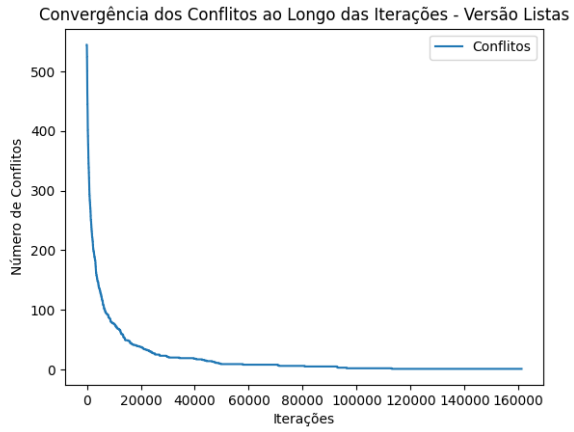

Figura 1: Convergência ao longo das iterações para a implementação com listas de controle.

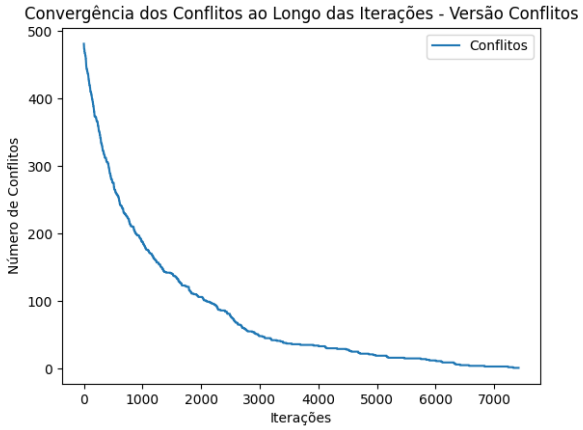

Figura 2: Convergência ao longo das iterações para a implementação com foco em rainhas em conflito.

Tabela I: Comparação de Desempenho entre as Implementações

Versão	Média	Mediana	Desvio Padrão	Melhor Caso	Pior Caso
Listas	19.52	6	45.78	0	545
Conflitos	90.60	57	101.01	0	467

V. Conclusões

A análise dos resultados nas Tabela 1 e Figuras 1 e 2 revela diferenças significativas na eficiência entre as duas implementações do algoritmo de Busca Tabu para o problema das N-Rainhas. A implementação final, que se concentra nas trocas entre rainhas em conflito, mostrou maior rapidez de convergência e desempenho superior em relação à versão inicial com listas de controle.

Os dados apontam que, enquanto a primeira implementação levou 450 segundos para resolver o problema com 1000 rainhas, a versão otimizada concluiu a tarefa em apenas 1,5 segundo. Esse avanço é atribuído à estratégia de geração de vizinhos, que evita trocas desnecessárias e foca nas rainhas que causam conflitos. Com isso, a análise de conflitos para a geração de vizinhos reduz a complexidade e melhora a eficiência geral do algoritmo.

Esses resultados destacam a importância de otimizar a geração de vizinhos em algoritmos de busca, demonstrando que a identificação e resolução de problemas específicos podem melhorar significativamente o desempenho.