Solução do Elo Maluco Utilizando Busca Heurística A*

Itallo Lobo Leite Carneiro
Engenharia de Computação
Universidade Federal de Itajubá - Campus Itabira
Itabira, Brasil

Pedro Fernandes Aguiar
Engenharia de Computação
Universidade Federal de Itajubá - Campus Itabira
Itabira, Brasil

***Resumo*—Área: Este trabalho se insere na área de Teoria dos Grafos e Otimização Combinatória, com foco em algoritmos de busca heurística.**

Problema: O Elo Maluco é um quebra-cabeça que apresenta desafios significativos de otimização combinatória, requerendo a determinação de uma sequência ótima de movimentos para sua resolução.

Proposta: Foi desenvolvida uma solução baseada no algoritmo A* com uma heurística especializada que considera padrões de cores e alturas das peças.

Resultados: A solução proposta demonstrou eficiência ao resolver Elo Maluco e cumpriu o objetivo final.

I. Introdução

A. Área

A otimização combinatória é um campo da matemática aplicada que busca encontrar a melhor solução possível para problemas que envolvem a seleção de um subconjunto de elementos de um conjunto finito, de acordo com algum critério de otimização. A teoria dos grafos, por sua vez, estuda as propriedades e aplicações de grafos, que são estruturas matemáticas usadas para modelar relações entre objetos. Ambas as áreas são fundamentais na resolução de quebra-cabeças computacionais, como o Elo Maluco.

B. Problema

O Elo Maluco consiste em um tabuleiro 4x4 com peças de diferentes cores e alturas que devem ser organizadas seguindo regras específicas. O desafio principal está em determinar a sequência mínima de movimentos necessários para atingir o estado objetivo. Este problema é complexo devido ao grande número de possíveis estados e à necessidade de considerar tanto a cor quanto a altura das peças. A dificuldade aumenta exponencialmente com o número de movimentos necessários para resolver o quebra-cabeça, tornando a busca por uma solução ótima um desafio significativo.

Os movimentos possíveis no quebra-cabeça Elo Maluco são descritos na Tabela I.

1) Descrição das Codificações: A Tabela II apresenta as codificações essenciais utilizadas no jogo Elo Maluco. A primeira parte define os códigos das cores (vm, vr, am, br) que identificam cada peça no tabuleiro. A segunda parte estabelece os códigos para as posições verticais (s, m, i) que determinam a altura de cada segmento do elo. Estas codificações são fundamentais para a representação do estado do jogo e para o processamento dos movimentos pelo algoritmo A*.

Tabela I: Movimentos possíveis no quebra-cabeça Elo Maluco

Código	Descrição
rsd	Rotacionar a parte superior para a direita
rse	Rotacionar a parte superior para a esquerda
rid	Rotacionar a parte inferior para a direita
rie	Rotacionar a parte inferior para a esquerda
mfc	Mover face abaixo da vazia para cima
mfb	Mover face acima da vazia para baixo

Tabela II: Codificação do jogo

Codificação de Cores	
Cor	**Código**
Vermelho	vm
Verde	vr
Amarelo	am
Branco	br

Codificação das Partes do Elo	
Parte do Elo	**Código**
Superior	s
Intermediário	m
Inferior	i

C. Abordagem Proposta

Desenvolveu-se uma solução baseada no algoritmo A*, que é um algoritmo de busca heurística amplamente utilizado em problemas de otimização. A heurística especializada desenvolvida para este trabalho considera:

- **Distribuição de cores nas colunas:** Cada coluna do tabuleiro deve conter peças de uma única cor, exceto a coluna branca, que pode conter uma peça "vazia".
- **Padrões de altura das peças:** As peças em cada coluna devem seguir um padrão específico de alturas, como 's' (superior), 'ms' (meio superior), 'mi' (meio inferior) e 'i' (inferior).
- **Posição relativa entre peças:** A posição das peças deve ser tal que minimize a distância entre a configuração atual e a configuração objetivo.

Existêm duas funções Heurísticas no programa, a função heurística Simples e a Heurística usando A* $h(n)$.Heurística usando A* $h(n)$ avalia cada estado do quebra-cabeça consi-

derando:

$$h(n) = w_1 C(n) + w_2 H(n) + w_3 P(n) \quad (1)$$

Onde:

- $C(n)$ representa o custo relacionado à distribuição de cores
- $H(n)$ avalia os padrões de altura
- $P(n)$ considera a posição relativa das peças

Será detalhado melhor na Modelagem da Proposta.

Já a função Heurística Simples:

1) Função `calcularHeuristicaSimples`: A função `calcularHeuristicaSimples` é responsável por calcular uma heurística simples para guiar a busca do algoritmo A*. Esta heurística é utilizada para estimar o custo restante para alcançar o estado objetivo a partir do estado atual do tabuleiro. A seguir, é descrito o funcionamento detalhado da função.

a) Descrição da Heurística: A heurística simples calcula a soma das distâncias de Manhattan de cada peça em relação à sua posição objetivo. A distância de Manhattan entre duas posições (x_1, y_1) e (x_2, y_2) é definida como:

$$d_{Manhattan} = |x_1 - x_2| + |y_1 - y_2| \quad (2)$$

b) Cálculo da Heurística: Para cada peça no tabuleiro, a função `calcularHeuristicaSimples` determina a posição atual da peça e a posição objetivo. Em seguida, a distância de Manhattan entre essas duas posições é calculada e acumulada em uma variável que representa o custo heurístico total.

c) Exemplo de Cálculo: Considere um tabuleiro 4x4 onde a peça na posição $(0, 0)$ deve ser movida para a posição $(3, 3)$. A distância de Manhattan para esta peça seria:

$$d_{Manhattan} = |0 - 3| + |0 - 3| = 3 + 3 = 6 \quad (3)$$

Se houver múltiplas peças fora de suas posições objetivo, a função `calcularHeuristicaSimples` somará as distâncias de Manhattan de todas as peças para obter o custo heurístico total.

d) Importância da Heurística: A heurística simples é admissível, o que significa que nunca superestima o custo real para alcançar o estado objetivo. Isso é crucial para garantir que o algoritmo A* encontre a solução ótima. Embora seja uma heurística básica, ela é eficiente em termos de tempo de computação e pode ser suficiente para resolver muitos problemas de busca.

e) Conclusão: A função fornece uma maneira eficiente de estimar o custo restante para alcançar o estado objetivo, guiando a busca do algoritmo A* de forma A abordagem proposta envolve a modelagem do problema como um grafo, onde cada nó representa um estado do tabuleiro e cada aresta representa um movimento válido. O algoritmo A* é utilizado para explorar este grafo, guiado pela função heurística que estima o custo restante para alcançar o estado objetivo.

D. Estrutura do Artigo

O artigo está organizado da seguinte forma: a Seção 2 apresenta trabalhos relacionados, a Seção 3 detalha a solução proposta, a Seção 4 apresenta os resultados experimentais e a Seção 5 conclui o trabalho.

E. Modelagem da Proposta

A solução utiliza uma função heurística $h(n)$ que avalia cada estado do quebra-cabeça considerando:

$$h(n) = w_1 C(n) + w_2 H(n) + w_3 P(n) \quad (4)$$

Onde:

- $C(n)$ representa o custo relacionado à distribuição de cores
- $H(n)$ avalia os padrões de altura
- $P(n)$ considera a posição relativa das peças

A função heurística é projetada para ser admissível, garantindo que nunca superestime o custo real para alcançar o estado objetivo. Isso é crucial para a corretude do algoritmo A*, assegurando que a solução encontrada seja ótima.

*F. Algoritmo A**

O algoritmo A* é um algoritmo de busca que utiliza uma função de custo $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo do caminho do nó inicial até o nó n, e $h(n)$ é a estimativa do custo do nó n até o nó objetivo. O algoritmo expande os nós na ordem de menor custo $f(n)$, garantindo que o primeiro nó expandido que corresponde ao estado objetivo seja a solução ótima.

G. Implementação do Algoritmo

A implementação do algoritmo A* para o Elo Maluco envolve a definição de estados, movimentos válidos e a função heurística. Cada estado é representado por uma matriz 4x4, e os movimentos válidos incluem rotações e movimentos de peças. A função heurística é calculada com base na distribuição de cores, padrões de altura e posição relativa das peças.

1) Uso de Threads e Geração de Números Aleatórios: O algoritmo utiliza múltiplas threads para realizar a busca de forma eficiente. A seguir, é descrito o funcionamento detalhado do uso das threads e da geração de números aleatórios.

a) Inicialização: O algoritmo começa inicializando algumas variáveis globais, como `deveTerminar` e `melhorPontuacaoGlobal`. Em seguida, são lançadas múltiplas threads, cada uma executando a função `threadBusca`.

b) Lançamento das Threads: Para cada thread, a função `threadBusca` é chamada com a grade inicial do quebra-cabeça e o identificador da thread. As threads são armazenadas em um vetor para que possam ser aguardadas posteriormente.

c) Execução das Threads: Cada thread executa a função `threadBusca`, que realiza a busca pela solução ótima utilizando uma fila de prioridade . A fila de prioridade armazena os nós a serem explorados, ordenados pela prioridade calculada pela heurística.

d) Geração de Números Aleatórios: Dentro de cada thread, é utilizado um gerador de números aleatórios (`std::mt19937`) para determinar o número de movimentos de embaralhamento. Isso adiciona uma variação aleatória ao processo de busca, ajudando a explorar diferentes caminhos possíveis.

e) Busca Heurística: A busca é realizada em um loop que continua até que a fila de prioridade esteja vazia ou que a variável `deveTerminar` seja verdadeira. Em cada iteração, o nó com a menor prioridade é removido da fila e processado.

f) Atualização da Melhor Pontuação: Se a prioridade do nó atual for menor que a melhor pontuação global, a variável `melhorPontuacaoGlobal` é atualizada com segurança utilizando um mutex (`std::mutex`). Isso garante que a atualização seja feita de forma segura em um ambiente multithread.

g) Conclusão das Threads: Após todas as threads concluírem a busca, a thread principal aguarda a conclusão de todas as threads utilizando `thread.join()`. Em seguida, a melhor solução encontrada é exibida.

2) Descrição das Funções Principais:

- **threadBusca**: Esta função é responsável por realizar a busca heurística em cada thread. Ela utiliza uma fila de prioridade para armazenar os nós a serem explorados e um conjunto para armazenar os nós já explorados. A heurística é calculada para cada nó, e a melhor pontuação é atualizada de forma segura utilizando um mutex.
- **calcularHeuristica**: A função `calcularHeuristica` é utilizada para calcular a prioridade de cada nó, baseada na heurística definida. Esta função é crucial para guiar a busca na direção da solução ótima.

3) Conclusão: O uso de múltiplas threads permite que o algoritmo explore diferentes caminhos possíveis em paralelo, aumentando a eficiência da busca. A geração de números aleatórios adiciona variação ao processo, ajudando a evitar que a busca fique presa em caminhos subótimos. A atualização segura da melhor pontuação garante que a solução encontrada seja a melhor possível entre todas as threads.

4) Descrição das Funções Principais:

- **processGame(Elo& elo):** Esta função é responsável por processar o jogo. Ela lê o estado inicial do tabuleiro a partir de um arquivo XML, exibe o estado inicial, lê a sequência de ações a serem executadas, processa essas ações no tabuleiro e exibe o estado final do tabuleiro.
- **exibirGrade(const Grade& grade):** Esta função exibe o estado atual do tabuleiro no console. Ela percorre cada linha e coluna do tabuleiro e imprime os valores das peças.
- **processarAcoes(Grade& grade, const vector<string>& acoes):** Esta função processa uma sequência de ações no tabuleiro. Para cada ação na lista de ações, ela chama a função correspondente para realizar o movimento no tabuleiro.
- **salvarEstados(const string& filename, const Elo::Grade& estadoInicial, const vector<string>& acoes):** Esta função salva os estados iniciais e finais do tabuleiro em um arquivo XML. Ela cria um novo documento XML, adiciona os estados iniciais e finais, e salva o documento no arquivo especificado.

H. Bibliotecas Utilizadas

- `<iostream>`
- `<thread>`
- `<queue>`
- `<unorderedset>`
- `<vector>`
- `<algorithm>`
- `<cstdlib>`
- `<ctime>`
- `<sstream>`
- `<mutex>`
- `<conditionvariable>`
- `<chrono>`
- `<random>`
- `<stdlib.h>`
- `<stdio.h>`

II. Resultados Experimentais

Os experimentos foram conduzidos considerando diferentes níveis de complexidade:

- 25 movimentos: 1800 iterações, tempo médio de 1 segundo
- 50 movimentos: 10000 iterações, tempo médio de 2 segundos
- 100 movimentos: 111000 iterações, tempo médio de 4 segundos

O algoritmo paralelo com componente aleatório demonstrou:

- Maior robustez para casos complexos
- Redução significativa no tempo de processamento
- Melhor qualidade das soluções encontradas (menos movimentos)
- Escalabilidade quase linear até 8 threads

Os resultados mostram que a solução proposta é capaz de resolver qualquer Elo Maluco com qualquer posição de embaralhamento em um tempo razoável, encontrando soluções otimizadas com um número reduzido de movimentos.

III. Conclusões

A solução proposta demonstrou-se eficaz para Elo Maluco , encontrando soluções otimizadas com significativa redução no número de movimentos necessários. A abordagem baseada no algoritmo A* com uma heurística especializada mostrou-se eficiente e pode ser aplicada a outros problemas de otimização combinatória.

Referências

Zhang, Jing, et al. "Autonomous land vehicle path planning algorithm based on improved heuristic function of A-Star."International Journal of Advanced Robotic Systems 18.5 (2021): 17298814211042730.

Wang, Pengyu, et al. "Improved A-star algorithm based on multivariate fusion heuristic function for autonomous driving path planning."Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering 237.7 (2023): 1527-1542.

Giorgi, Roberto, and Alberto Scionti. "A scalable thread scheduling co-processor based on data-flow principles."Future Generation Computer Systems 53 (2015): 100-108.