Problema do deslocamento do cavalo utilizando Branch-and-Bound

Deslocamento mínimo do Cavalo em um tabuleiro de xadrez NxN

1st Allef Fernandes Santos
dept. Engenharia de Computação
Universidade Tecnológica Federal do Paraná *UTFPR*
Pato Branco,Brasil
allefs@alunos.utfpr.edu.br

2nd Guilherme Macedo Baggio
dept. Engenharia de Computação
Universidade Tecnológica Federal do Paraná *UTFPR*
Pato Branco,Brasil
guilhermebaggio@alunos.utfpr.edu.br

3nd Lucas Arruda Silva
dept. Engenharia de Computação
Universidade Tecnológica Federal do Paraná *UTFPR*
Pato Branco, Brasil
lucasarruda@alunos.utfpr.edu.br

4nd Lucas Tarcisio Moraes Pies
dept. Engenharia de Computação
Universidade Tecnológica Federal do Paraná *UTFPR*
Pato Branco,Brasil
lucaspies@alunos.utfpr.edu.br

5nd Yuri Gabriel dos Reis Souza
dept. Engenharia de Computação
Universidade Tecnológica Federal do Paraná *UTFPR*
Pato Branco,Brasil
yurigabriel@alunos.utfpr.edu.br

***Resumo*—Este trabalho aborda o problema do Deslocamento do Cavalo em um tabuleiro de xadrez *NxN*. O objetivo principal é encontrar o caminho mais eficiente para que o cavalo se desloque de uma coordenada inicial (*x0, y0*) até uma coordenada final (*xf, yf*), levando em consideração a movimentação do cavalo pelo tabuleiro que segue as regras do xadrez. Para resolução da problemática, foi implementado um algoritmo em C, utilizando *Branch-and-Bound* como paradigma de programação. No presente documento, foi discutido o método utilizado, bem como a comparação com outras técnicas de desenvolvimento, análise de complexidade e apresentação dos resultados obtidos.**

***Keywords*—Passeio do cavalo, Algoritmo, Xadrez, Complexidade, Matriz, Branch-and -bound**

I. Introdução

O xadrez é um jogo de longa data, que atrai jogadores do mundo inteiro pela sua complexidade. Dentre as inúmeras peças presentes no jogo, o cavalo é a que possui o movimento mais único, realizando deslocamentos em "L". Esse movimento não linear, faz com que a previsão e o cálculo dos movimentos do cavalo sejam uma tarefa desafiadora.

O objetivo é garantir uma eficiência computacional para a resolução do problema, utilizando um algoritmo que permite encontrar a melhor solução para tabuleiros de diversas dimensões. A linguagem de programação escolhida para implementar a solução foi a linguagem C.

A escolha deste problema se deu pela sua aplicabilidade prática em diferentes contextos. O movimento único do cavalo no xadrez desperta o interesse por sua capacidade de modelar desafios complexos de forma intuitiva. Além disso, sua resolução pode ser usada para problemas reais como, logística, jogos, robótica e otimização.

II. Descrição do problema e motivações para a escolha das estratégias

A. Descrição do problema

O problema do Deslocamento do cavalo consiste em determinar o menor número de movimentos necessários para que a peça do cavalo em um tabuleiro de xadrez de dimensões $N \times N$ se desloque de uma posição inicial (x_0,y_0) até uma posição final (x_f,y_f). O cavalo tem um movimento em "L", que consiste em mover duas casas em uma direção (vertical ou horizontal) e, em seguida, uma casa em direção perpendicular. Para atingir o objetivo foi necessário estruturar o problema para facilitar a sua solução, que segue a seguinte estrutura:

Entrada:

- Dimensão do tabuleiro *N;*
- Posição inicial (x_0,y_0);
- Posição final (x_f,y_f).

Saída:

- Menor número de movimentos necessários para alcançar a posição final a partir da posição inicial;
- Caminho percorrido pelo cavalo mostrando os passos no tabuleiro.

A solução deste problema exige uma abordagem eficiente, que explore o espaço de busca de forma sistemática e otimizada. Para isso, utilizamos a técnica de *Branch-and-Bound*, que permite limitar a exploração de movimentos, garantindo que apenas as soluções promissoras sejam consideradas, reduzindo assim o tempo de execução e o uso de recursos computacionais.

B. Motivações para a escolha da estratégia:

*Branch-and-Boun*d é uma técnica de otimização que explora sistematicamente o espaço de busca e descarta (poda) partes do espaço que não podem conter uma solução melhor do que a já encontrada. No contexto do problema do cavalo, essa técnica ajuda a limitar a exploração de movimentos, garantindo que apenas as soluções promissoras sejam

consideradas. A escolha do método para solucionar o problema foi motivada por várias considerações, técnicas e práticas. Em seguida serão discutidas as razões que tornam essa técnica particularmente adequada para este tipo de problema.

O problema do cavalo que envolve o cálculo do menor número de movimentos, é um exemplo clássico de problema de busca em um espaço de estados. O espaço de busca cresce exponencialmente com o tamanho do tabuleiro, tornando a enumeração completa de todas as possíveis sequências impraticável para tabuleiros muito grandes. O método *Branch-and-Bound* é ideal para esse cenário pois reduz o espaço de busca ao podar ramos que não podem levar a uma solução melhor do que a já encontrada, reduzindo significativamente o número de estados que precisam ser explorados, além de priorizar a exploração de estados que são mais promissores, baseando-se nos limitantes calculados, levando a uma busca mais direcionada e eficiente.

C. Comparação com outras técnicas:

Backtracking: Explora todas as possibilidades de maneira sistemática, retrocedendo quando uma solução parcial não pode ser completada, embora seja simples de implementar, tal técnica pode ser extremamente ineficiente para problemas com grandes espaços de busca pois não tem informações que limitem a busca. Desse modo o *Branch-and-Bound* melhora a eficiência do *Backtracking* ao incorporar limitantes, evitando a exploração de sub-árvores inteiras que não podem melhorar a solução atual.

Método Guloso: Faz a escolha que parece ser a melhor no momento sem considerar as implicações futuras, não garante uma solução ótima pois, as decisões locais não necessariamente levam à solução globalmente ótima. Logo *Branch-and-Bound* garante a obtenção da solução ótima global, enquanto o método guloso pode falhar em encontrar a melhor solução devido à sua natureza.

Divisão e Conquista: Divide o problema em subproblemas menores, resolve cada um independentemente e combina as soluções, não se aplica naturalmente a problemas de caminho mínimo, onde a interdependência entre subproblemas é alta. *Branch-and-Bound* é mais adequado para problemas de caminho mínimo pois, explora todas as possibilidades de maneira controlada e eficiente.

Assim, a escolha do método *Branch-and-Bound* para resolver o problema do cavalo, é justificada pela sua eficiência em lidar com grandes espaços de busca e sua capacidade de garantir a solução ótima. Comparado com outras técnicas, oferece um equilíbrio ideal entre a simplicidade da implementação e a eficiência computacional, tornando-o a estratégia mais adequada para esse tipo de problema.

III. SOLUÇÃO DO PROBLEMA

O tabuleiro de xadrez é representado por uma matriz *NxN*, onde cada posição é identificada por um par ordenado (*x, y*). O cavalo possui até oito movimentos possíveis a partir de uma posição dada, sendo eles:

- $(x+2, y+1)$
- $(x+2, y-1)$
- $(x-2, y+1)$
- $(x-2, y-1)$
- $(x+1, y+2)$
- $(x+1, y-2)$
- $(x-1, y+2)$
- $(x-1, y-2)$

Para que um movimento seja válido, a posição resultante deve estar dentro dos limites do tabuleiro, ou seja:

```
INÍCIO
1 Função ehValida(x, y, tamanhoDoTabuleiro)
2       Retorna verdadeiro se x >= 0 e x < tamanhoDoTabuleiro e
        y >= 0 e y <tamanhoDoTabuleiro;
FIM
```

Uma vez definido a movimentação e validado as posições, o método *Branch-and-Bound* pode ser implementado, sendo representado em partes a seguir:

A. Inicialização:

O algoritmo começa verificando se a posição inicial e a posição final estão dentro dos limites válidos do tabuleiro *NxN,* em seguida inicializa uma matriz "visitado" para marcar as posições já visitadas pelo cavalo e cria o "nó raiz" com a posição inicial e custo zero, inserindo-o em uma fila de nós a serem explorados.

```
INÍCIO
1 Função criarNo(x, y, numeroDePassos, parent)
2       Alocar memória para um novo nó (tree);
3       tree->pos.x ⬅ x;
4       tree->pos.y ⬅ y;
5       tree->numeroDePassos ⬅ numeroDePassos;
6       tree->parent ⬅ parent;
7       Retorna nó (tree)
FIM
```

B. Exploração das soluções:

Utilizando uma estratégia de fila de prioridade, o algoritmo explora cada nó removendo-o da fila de acordo com seu custo acumulado. Para cada nó explorado, verifica se ele corresponde à posição final desejada, se for aceito, o caminho até esse ponto é registrado, caso contrário, para cada movimento válido do cavalo a partir da posição atual, calcula-se uma estimativa do custo mínimo para chegar à posição final (limitante inferior).

O limitante inferior é uma estimativa do menor custo possível para alcançar a posição final (*xf,yf*) a partir de uma posição atual. Ele serve como uma heurística que ajuda o algoritmo a decidir quais ramos da árvore de busca devem ser explorados com prioridade, reduzindo o número de caminhos a serem analisados. No contexto do problema do Cavalo no Xadrez, uma abordagem comum para calcular o limitante inferior é através da distância de *Manhattan* entre a posição atual do cavalo e a posição final desejada. A distância de *Manhattan* é calculada somando as diferenças absolutas das coordenadas x e y entre as duas posições:

$$lim\_inf = abs(fim.x - novoX) + abs(fim.y - novoY)$$

Onde “*fim.x” e “fim.y”* representam coordenadas *x* e *y* da posição final, analogamente *“novoX” e “novoY”* são coordenadas *x* e *y* da nova posição que o cavalo poderia ocupar após o movimento.

Após calcular o limitante inferior para uma nova posição (*novoX, novoY*), o algoritmo compara o custo acumulado até o momento (*custo + 1*) somado ao limitante inferior com o limitante superior, que é o melhor custo encontrado até o momento, se essa soma for menor do que o limitante superior, significa que é promissor explorar essa nova posição pois, ela pode levar a um caminho mais curto até o destino final.

Se o custo acumulado mais o limitante inferior for menor do que o melhor custo encontrado até o momento (limitante superior), o novo nó é adicionado a fila para exploração.

O limitante superior é uma métrica que representa o menor custo encontrado até o momento para alcançar a posição final desejada, ele serve como um “*benchmark*” ou referência para comparar os custos acumulados de novos caminhos explorados durante a busca. Pode-se utilizar a constante “*INT_MAX”* da biblioteca “*limits.h”* para representar o maior inteiro possível. Isso garante que qualquer custo encontrado durante a busca será menor do que o limitante superior inicialmente, permitindo que o algoritmo atualize esse valor conforme encontra caminhos mais curtos. Durante a busca, sempre que um caminho completo do início ao fim é encontrado, o algoritmo verifica se o custo desse caminho é menor do que o limitante superior atual. Se for, o limitante superior é atualizado com esse novo custo e o caminho encontrado é registrado.

O limitante superior desempenha um papel fundamental na poda de ramos da árvore de busca. Ele permite que o algoritmo elimine os caminhos que não têm potencial para serem soluções ótimas (prioriza caminhos mais curtos). Dessa forma, quando um caminho completo é encontrado com um custo maior do que o limitante superior atual, ele é prontamente eliminado, portanto, reduz significativamente o espaço de busca, melhorando a eficiência computacional do algoritmo.

C. *Critério de parada:*

O algoritmo explora nós na fila até que todos sejam processados ou não haja mais nós promissores para explorar, realizando um corte por limitante superior. Se um nó é promissor, ele é expandido e seus filhos são adicionados à fila, caso contrário, o nó é descartado, evitando caminhos que não levam à solução ótima.

IV. Análise de complexidade tempo e de espaço de cada solução

A. *Complexidade de tempo*

A análise da complexidade de tempo envolve examinar as operações críticas e loops, no caso, o que mais influencia a complexidade desse código, se encontra na função “menorCaminho”, principalmente quando há a criação e inicialização da matriz “visitado”, como também, no processamento da fila para determinar o número total de operações.

Para a matriz “visitado”, a criação e inicialização da matriz envolvem dois loops aninhados, que são responsáveis por grande parte da complexidade de tempo da função. No loop externo, (i) é executado n vezes, e no loop interno (j) também é executado n vezes, onde n é o tamanho do tabuleiro. Portanto, a complexidade de tempo para criar e inicializar a matriz “visitada” é $O(n^2)$.

Na parte do processamento da fila, a análise é a seguinte, o loop “while” no pior caso, processa cada posição do tabuleiro uma vez, portanto, executa $O(n^2)$ vezes. Dentro do loop while, há um loop “for” que executa 8 vezes para cada movimento possível do cavalo, resultando em uma complexidade de $O(n^2) \times O(8) = O(n^2)$. Somando as complexidades das duas principais partes da função “menorCaminho”, temos $O(n^2)+O(n^2) = O(n^2)$. Portanto, a complexidade de tempo total da função “menorCaminho” é $O(n^2)$.

B. *Complexidade de Espaço*

Para o cálculo da complexidade de espaço seguindo o que foi relatado para o cálculo da complexidade de tempo, levando em conta as principais estruturas de dados que afetam o uso da memória como o tamanho da matriz *N x N,* matriz de visitados, função para criar o tabuleiro, função para imprimir o caminho e também a função menor caminho, temos para cada uma dessas uma complexidade de $O(n^2)$. A análise considera as principais funções e estruturas de dados que contribuem para o uso de memória, resultando em uma complexidade de espaço quadrática.

V. Resultados obtidos

Como resultados obtidos, segue a execução do código para diferentes tamanhos de tabuleiro, mostrando o menor caminho percorrido pelo cavalo para se deslocar da posição inicial até a final fornecida.

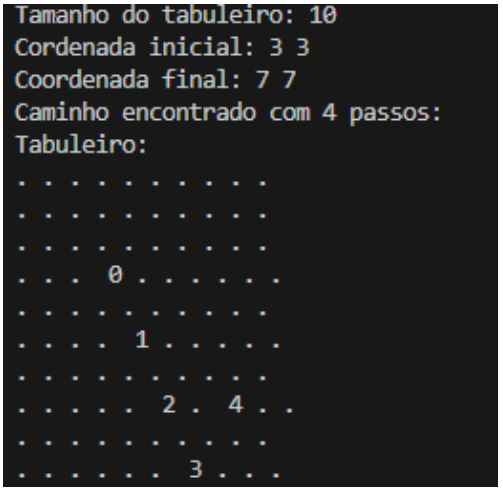

Figura 1: exemplo de saída com o caminho mínimo para um tabuleiro 10x10.

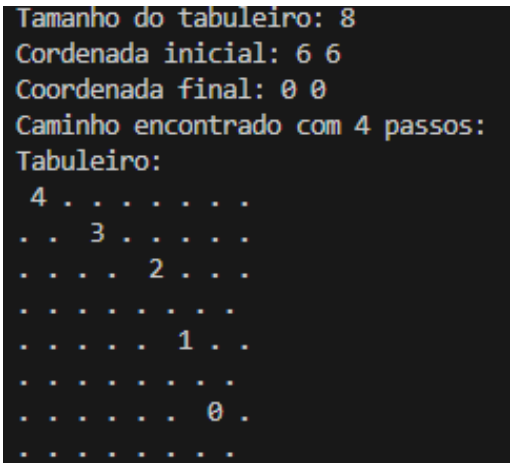

Figura 2: exemplo de saída com o caminho mínimo para um tabuleiro 8x8.

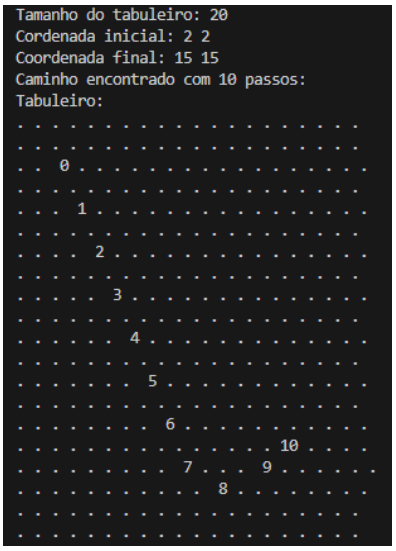

Figura 3: exemplo de saída com o caminho mínimo para um tabuleiro 20x20.

VI. Conclusão

A implementação do algoritmo foi abordada com sucesso nesse trabalho utilizando a técnica *Branch-and-Bound* para encontrar o caminho mais eficiente entre as posições inicial e final. A escolha da técnica se mostrou adequada devido a sua capacidade de explorar o espaço de busca de caminhos de maneira otimizada, priorizando percursos promissores.

Em suma, a solução proposta não só alcançou o objetivo de encontrar o menor caminho para o cavalo no tabuleiro de xadrez, mas também demonstrou a viabilidade e eficiência do uso do algoritmo para problemas similares. A aplicação prática desse algoritmo pode ser estendida para tabuleiros maiores ou diferentes configurações de jogo, mostrando sua versatilidade e robustez na resolução de problemas complexos de busca.

VII. Referências

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

[2] Pohl, I. (1971). Heuristic search viewed as path finding in a graph. Artificial Intelligence, 1(3-4), 193-204.

[3] Instituto de Matemática e Estatística, Universidade de São Paulo. "Grafos." Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/graphs.html.

[4] CLAUSEN, J. Branch and bound algorithms-principles and examples. Department of Computer Science, University of Copenhagen, 1999.

[5] Oliva, J. T Arquivos. AE43CP - Algoritmos e Estrutura de Dados II. Aula12. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2024.

[6] Linguagem C. "Arquivos em C – categoria usando arquivos." Disponível em: https://linguagemc.com.br/arquivos-em-c-categoria-usando-arquivos/.

VIII. Declaração de autoria

Allef Fernandes Santos: Contribuiu com a revisão da documentação do código, introdução, descrição do problema, solução do problema, formatação e referências. Contribuiu documentando o código e o deixando mais claro. Contribuiu com o desenvolvimento do README.

Lucas Arruda Silva: Como líder, propôs a divisão do grupo em subgrupos, a fim de distribuir as tarefas igualmente entre os membros. Contribuiu com a implementação do código e do método *Branch-and-Bound.* Também participou na elaboração do relatório com ênfase na descrição e solução do problema, resultados obtidos e conclusão, bem como, auxiliou nas demais tarefas quando necessário.

Lucas Tarcisio Morais Pies: Contribuiu na implementação e lógica da criação da matriz utilizada como tabuleiro e na sua impressão; revisou a documentação do código e aprimorou a lógica do encapsulamento das funções para melhor reusabilidade no código. Além disso, colaborou na elaboração da

introdução e na descrição do problema no relatório, assim como auxiliou no fornecimento das figuras no relatório.

Guilherme Macedo Baggio: Contribuiu com a revisão do código e do pseudocódigo,ajudou no versionamento e controle de versão do código, ficou responsável pela implementação da leitura de dados via .txt e com o cálculo das complexidades de tempo e espaço. Também contribuiu na documentação do código.

Yuri Gabriel dos Reis Souza: Contribuiu com a implementação do código que realiza o movimento do cavalo no tabuleiro, bem como parte do cálculo do mínimo de passos. Também realizou parte da explicação/documentação do código. Elaborou os pseudocódigos utilizados, assim como, resumo e revisão dos códigos e do relatório.