Problema do Cavalo no Xadrez

Trabalho de Algoritimos e Estrutura de Dados 2

1st Caetano Chinarelli Souza *Regitro Acadêmico 2344955*
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
2nd Guilherme Iago Marcante Della Libera *Regitro Acadêmico 2199572*
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
3rd Kelvyn Augusto Waltrick Nonato *Regitro Acadêmico 2345048*
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
4th Luiz Eduardo Rufatto *Regitro Acadêmico 2079933*
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
5th Vinicius Soares do Rosario *Regitro Acadêmico 2247305*
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná

I. Introdução

Com o tempo, a programação foi se desenvolvendo cada vez mais rápido ao redor do mundo, e com isso foi se criando das mais diversas técnicas para a resolução de muitos problemas existente, e com isso conseguindo cada vez mais a otimização de códigos para seu melhor funcionamento e execução.

No problema que foi escolhido para resolvermos, o deslocamento de um cavalo em um tabuleiro de xadrez, em que é necessário achar a melhor combinação de passos possíveis para chegar até a posição final definida pelo jogador. Com isso, se encaixa a otimização de resultados, fazendo com que o processamento seja cada vez menor, buscando o melhor caso entre todos os possíveis.

Ao fazer a escolha do melhor modelo para resolver o problema selecionado, foi analisado qual seria a ideia para melhor aproveitamento de memória e processamento, onde o backtraking acabou se destacando por ser um modelo que descarta soluções mediana ou péssimas sem mesmo terminar sua checagem, e escolhendo sempre o melhor caso para a resolução do problema.

Após o problema resolvido e a concretização da melhor ação a ser tomada, a possibilidade da resolução de outros tipos de problema desse meio utilizando essa mesma técnica poderá ser feita com muito mais rapidez e economizando memória e processamento da máquina que está sendo utilizada.

II. Problemática

A. Descrição do problema

O problema tratado foi o do deslocamento do cavalo no tabuleiro de xadrez. Esta proposta, segue a seguinte descrição (conforme consta nas especificações do trabalho): dado um tabuleiro de ordem N (NxN) e uma posição inicial (X0, Y0), o algoritmo deve encontrar uma solução ótima (caso exista) com a menor quantidade de movimentos para chegar a posição final. O padrão de movimento do cavalo em um jogo de xadrez normal deve ser seguido, isto é, "em L". Além disso, deve ser impresso a quantidade de passos e uma matriz onde cada elemento indica o número de passos para chegar à posição em questão.

A escolha deste problema resumiu-se a um acordo comum entre todos os integrantes do grupo através de uma votação, levando em consideração o consenso entre os membros que este foi de fácil compreensão, o que permitiu um planejamento mais claro e eficiente quanto a como abordá-lo.

B. Motivação para a escolha do problema

Para o desafio que nos propusemos a desenvolver, após alguns testes realizados, decidimos pela utilização do uso de backtracking, pois com ele sempre pegando o melhor caso sempre conseguir uma eficiência máxima.

E muitas vezes quando atrelado a programação dinâmica, é possível conseguir resultados melhor que o esperado, pois ele sempre evita vários cálculos desnecessários em diversos códigos, assim economizando ainda mais memória sempre que possível.

C. Estrategias para a solução

Do mesmo modo que são tratados diversos problemas envolvendo algoritmos, é possível encontrar múltiplas soluções através de diferentes estratégias aplicadas. O critério de desempate, neste caso e em geral, reduz-se a escolha do algoritmo com melhor eficiência em comparação aos demais, ou seja, que possui o menor custo computacional. Ainda para a escolha, consideram-se a facilidade de manipular a estratégia para a aplicação desejada e a legibilidade do código. A partir disso, foi decidido a escolha de duas estratégias de algoritmos comprovadamente eficientes que nos auxiliam com a resolução do cenário em questão: backtracking aliada a busca em largura.

Backtracking é um tipo de algoritmo que deriva da busca por força bruta, sendo um refinamento desta. A estratégia em questão é utilizada para encontrar todas ou algumas soluções para problemas computacionais, especialmente problemas de satisfação de restrições. Consiste em construir candidatos para as soluções, "abandonando" um candidato (origina o termo backtrack, traduzindo, "retroceder") ao determinar que não pode ser completado para uma solução válida. Isto implica que o algoritmo só é válido para problemas que admitem "soluções parciais" como candidatos para soluções, então possibilitando um teste de validez relativamente simples para que a solução possa ser completada.

Conceitualmente, backtracking é frequentemente representado por uma estrutura de árvore, seguindo o padrão de busca em profundidade. Os candidatos a solução são os nós da árvore e cada nó de solução parcial é nó pai dos nós candidatos a solução, portanto também são nós intermediários. Se um nó candidato parcial é determinado como inválido, toda a subárvore desse nó também é desconsiderada, então a busca retrocede para o nó anterior e repete o processo de verificação para os demais nós da estrutura. A árvore é percorrida recursivamente, garantindo que toda a árvore válida é percorrida, mas não toda a árvore potencial (que contém todos os possíveis candidatos mas nem todos válidos).

O termo backtrack é creditado a Derrick H. Lehmer na década de 1950, porém diversos autores trabalharam acerca do tema ao longo dos anos.

Já a busca em largura é um algoritmo de busca que expande e examina todos os vértices de um grafo ou todos os nós de uma árvore, dependendo da representação escolhida. Na prática, o algoritmo geralmente utiliza uma estrutura de fila para marcar os nós visitados e os nós ainda não visitados. Dessa maneira, percorre todas as alternativas em um determinado nível até encontrar a solução ótima. Portanto, a busca em largura provou-se um algoritmo eficiente e apropriado para a resolução do problema do menor caminho para o cavalo no tabuleiro de tamanho N, dado que, por sua natureza, garante que o menor caminho seja escolhido.

A busca em largura e sua aplicação em encontrar componentes conectados de grafos foi inventado pelo engenheiro e cientista da computação alemão Konrad Zuse em 1945, porém seu estudo foi publicado apenas em 1972. Foi reinventado em 1959 pelo também cientista da computação e professor de matemática americano Edward F. Moore, que a utilizou para encontrar o menor caminho em um labirinto.

Para a resolução do problema, utilizamos a busca em largura para encontrar o menor caminho que o cavalo deve percorrer até chegar ao destino e ainda uma estrutura auxiliar (matriz "pai") para registrar e posteriormente reconstruir o caminho percorrido. A reconstrução é feita através de backtracking, utilizando a estrutura citada que mantém o pai de cada célula de posição, por meio de uma função recursiva que preenche uma outra matriz com os passos do cavalo. Ela percorre o caminho do destino de volta à origem, utilizando uma abordagem recursiva para reconstruir o percurso. Essa técnica permite explorar todas as possíveis opções para construir o caminho correto, mas neste caso específico, a função apenas segue as informações previamente calculadas pela busca em largura, o que simplifica o processo de backtracking.

III. Descrição das Soluções do Problema

Tendo que o problema que tivemos em mãos foi o cavalo que se move em padrões pré-definidos por um tabuleiro de xadrez, do local de saída escolhido pelo usuario, até um local de chegada escolhido pelo usuario, sendo o tabuleiro alocado dentro de uma matriz NxN, procuramos por um meio de encontrar o menor caminho que leva ao local de chegada, um meio de marcá-lo, e de mostrá-lo em uma matriz.

A solução encontrada para a resolução do problema foi uma combinação entre BFS e backtracking. A busca em profundidade do BFS encontra todos os caminhos possíveis para a área escolhida, fazendo diversos caminhos e os marcando para calcular cada caminho possível, salvando sempre a distância de cada local encontrado até a origem em seus próprios locais de armazenamento, isto é, se o caminho for encontrado, caso contrário, e retornado o valor –1. A realização do backtracking e enfim usada nos resultados encontrados pela aplicação anterior, depois de conferido o menor caminho, para então seguir da célula final a célula inicial, fazendo o caminho reverso ao qual foi dado do menor caminho, marcando assim na matriz o caminho que deve ser impresso no tabuleiro, e finalizando o processo temos uma função para a impressão da matriz no formato informado pelo usuario para ser o tabuleiro de xadrez.

IV. Análise de Complexidade Tempo e de Espaço

Para a analise de complexidade foi calculada em partes para maior compreensão de das características do codigo de formas a chegarmos as melhores resultados, dado as características do problemas e suas soluções.

A. ehValido

```
int ehValido(int x, int y, int n) {
     return (x >= 0 && x < n && y >= 0 && y < n);
}
```

Está é uma função que faz uma comparação para determinar se o argumento passado é válido.

A complexidade de tempo é constante, O(1), por conta que a função apenas faz uma comparação aritmética.

A complexidade de espaço é constante, O(1), por conta que ela apenas usa quantidades constantes de memória em seu armazenamento, apenas para guardar variáveis temporárias utilizadas na comparação.

B. BFS

```
int bfs(int n, int iniX, int iniY, int fimX, int fimY, int caminho[n][n], Cell pai[n][n]) {

    int visitou[n][n];
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            visitou[i][j] = 0;
            pai[i][j] = (Cell){-1, -1, -1};
        }

    Cell fila[MAXN * MAXN];
    int frente = 0, tras = 0;

    fila[tras++] = (Cell){iniX, iniY, 0};
    visitou[iniX][iniY] = 1;

    while (frente < tras) {
        Cell cell = fila[frente++];

        if (cell.x == fimX && cell.y == fimY) {
            return cell.dist;
        }

        for (int i = 0; i < 8; i++) {
            int x = cell.x + dx[i];
            int y = cell.y + dy[i];

            if (ehValido(x, y, n) && !visitou[x][y]) {
                visitou[x][y] = 1;
                pai[x][y] = (Cell){cell.x, cell.y, cell.dist + 1};
                fila[tras++] = (Cell){x, y, cell.dist + 1};
            }
        }
    }
    return -1;
}
```

Está é uma função de Busca em Largura (BFS).

A complexidade de tempo dessa função é de O(n²) por causa que temos interações de matrizes NxN, assim como laços for NxN, tudo isso vai nos resultar em uma complexidade NxN, ou seja, O(n²). Ao todo, o algoritmo BFS visita cada célula no pior caso uma única vez, isso vai levar ao todo O(n²) operações para fazer toda a visita.

A complexidade de espaço dessa função é O(n²) por conta de usarmos uma matriz auxiliar NxN para manter guardado valores das células visitadas, as quais são as matrizes dos caminhos do cavalo, a visitou, ela vai ocupar O(n²) de espaço. isso nos resulta em uma complexidade de espaço NxN, ou seja, O(n²).

C. *backtrackCam*

```
void backtrackCam(int n, int iniX, int iniY, int fimX, int fimY, int caminho[n][n], Cell pai[n][n]) {
    if (fimX == iniX && fimY == iniY) {
        caminho[iniX][iniY] = 0;
        return;
    }

    Cell p = pai[fimX][fimY];
    caminho[fimX][fimY] = p.dist;
    backtrackCam(n, iniX, iniY, p.x, p.y, caminho, pai);
}
```

Está é uma função de Backtracking.

A complexidade de tempo dessa função é de O(n²), por conta que ela usa uma chamada recursiva para colocar cada célula do caminho do final até a cell do início, visitando cada uma delas. A profundidade máxima da recursão é limitada pela quantias de células que estão pelo caminho, teríamos um pior caso de NxN, a matriz toda, ou seja, O(n²).

A complexidade de espaço é de O(n²) por conta que usamos espaço na pilha proporcionalmente à profundidade máxima da recursão, a qual é NxN. Também temos as matrizes de tamanho NxN, ou seja, o espaço ocupado por elas é O(n²).

D. *imprimeTab*

```
void imprimeTab(int n, int caminho[n][n], int x, int y) {
    printf("    ");
    for (int j = 0; j < n; j++) {
        printf("%d:", j);
        printf(" ");
    }
    printf("\n");

    for (int i = 0; i < n; i++) {
        printf("%d: ", i);
        for (int j = 0; j < n; j++) {
            if(i == x && j == y){
                printf("%2d*", caminho[i][j]);
            }
            else{
                printf("%2d ", caminho[i][j]);
            }
        }
        printf("\n");
    }
}
```

A complexidade de tempo é de O(n²). Temos dois laços for utilizados para imprimir a matriz, os dois seguindo até N, dentro deles temos operações printf simples que são constantes, ao todo, sendo denominada O(n²) pelo seu loop de impressão NxN.

A complexidade de espaço é O(1). As variáveis que utilizamos dentro dela são apenas constantes, e suas variáveis criadas são constantes i e j. O resto passamos para dentro da função, sem alocação dentro dela, logo utilizamos as variáveis locais e espaço de pilha utilizado durante sua execução, logo temos O(1).

E. *Main*

```
int main() {
    int n;
    scanf("%d", &n);

    int iniX, iniY, fimX, fimY;
    scanf("%d %d", &iniX, &iniY);
    scanf("%d %d", &fimX, &fimY);

    int caminho[n][n];
```

```c
    Cell pai[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            caminho[i][j] = 0;

    int result = bfs(n, iniX, iniY, fimX, fimY, caminho, pai);
    if (result != -1) {
        backtrackCam(n, iniX, iniY, fimX, fimY, caminho, pai);
        printf("Menor caminho encontrado (passos): %d\n", result);
        imprimeTab(n, caminho, iniX, iniY);
    } else {
        printf("Nao ha caminho possivel para o cavalo.\n");
    }

    return 0;
}
```

A complexidade de tempo dentro da Main é $O(n^2)$ por conta que temos um laço for de NxN para criação da matriz utilizada no algoritmo, assim como também temos que a função BFS e backtrackCam tem a complexidade de $O(n^2)$ em seu pior caso.

A complexidade de espaço dentro da Main é de $O(n^2)$ por conta que temos duas matrizes de duas dimensões, a matriz caminho e pai, ambas são NxN. Essas matrizes armazenam informação sobre o caminho e suas células parentes durante o algoritmo de procura. Por conta de seu tamanho NxN, elas ocupam espaço $O(n^2)$.

F. Conclusão

Em geral, temos que a complexidade total do algoritmo seria:

- Complexidade de Tempo: $O(n^2)$.
- Complexidade de Espaço: $O(n^2)$.

Nos garantindo um resultado otimizado e esperado, na medida da lógica dos algoritmos utilizados para a solução do problema.

V. Conclusão

Concluímos que a complexidade de tempo e espaço do algoritmo utilizado para solução do problema do cavalo foi de $O(n^2)$.

Chegamos a um bom resultado, visando que tanto os algoritmos de BFS e Backtracking utilizados para a resolução do problema haviam em seu pior caso $O(n^2)$, conseguimos manter o algoritmo tendo sua complexidade quadrática, se tornando uma solução viável para entregarmos um bom resultado.

Quando consideramos que o tamanho do tabuleiro é NxN, onde N seria os números de células, Isso nos resulta um desempenho que é proporcional a n^2, o que reflete para a gente sua complexidade tanto em tempo quanto espaço de $O(n^2)$, isso confirma que a complexidade

VI. Declaração de Autoria

O integrante Caetano C. Souza foi responsável pelo desenvolvimento do código, o que inclui implementação e correção das funções de busca em largura para o menor caminho do cavalo, backtracking para reconstrução e representação do caminho, impressão da matriz e correções na main. O integrante ainda foi responsável pela documentação do código, a descrição do problema, motivações para a escolha das estratégias e sua descrição no artigo.

O integrante Guilherme I. M. D. Libera, foi responsável pela elaboração de parte do artigo como introdução do tema e a motivação da escolha em específico, e demais partes do desenvolvimento quando necessário. E a partir do artigo, a elaboração da apresentação de defesa do artigo final feita presencialmente.

O integrante Kelvyn A. W. Nonato foi responsável pelo cálculo de complexidade de tempo e espaço do algoritmo dentro de cada função específica e fazer a conclusão dos resultados obtidos na forma de criação, aprimoramento e cálculo de complexidade do código. O integrante também auxiliou na observação de erros dentro do código e documentando a conclusão do artigo.

O integrante Luiz Eduardo Rufatto foi responsável pela primeira elaboração do código, a função qual foi repassada para o aluno Caetano posteriormente, o relatório sobre ”Descrição da solução do problema” e ajudas em discussões sobre pormenores e ajustes de código.

O integrante Vinícius S. Rosário foi responsável pela criação da base do código, bem como pela implementação dos algoritmos de backtracking, BFS e o arquivo de entrada de dados, contribuiu para a construção do relatório, fornecendo informações relevantes e detalhes sobre a implementação do código, organizou o repositório Git, criou o README e garanti que o código-fonte estivesse adequadamente versionado e assumiu a responsabilidade de organizar o grupo, distribuindo tarefas entre os membros e garantindo uma colaboração eficiente.

Referências

[1] Zuse, Konrad (1972), Der Plankalkül (in German), Konrad Zuse Internet Archive. See pp. 96–105 of the linked pdf file (internal numbering 2.47–2.56).

[2] Moore, Edward F. (1959). ”The shortest path through a maze”. Proceedings of the International Symposium on the Theory of Switching. Harvard University Press. pp. 285–292. As cited by Cormen, Leiserson, Rivest, and Stein.

[3] CORMEN, T. et al. Algoritmos: Teoria e Prática. Editora Campus, 2012.

[4] “Breadth First Search or BFS for a Graph - GeeksforGeeks”. GeeksforGeeks. Consult. 2024-06-23. [Em linha]. Disponível: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[5] “Shortest path in an unweighted graph - GeeksforGeeks”. GeeksforGeeks. Consult. 2024-06-23. [Em linha]. Disponível: https://www.geeksforgeeks.org/shortest-path-unweighted-graph/

[6] “Minimum Steps to Reach the Target - GeeksforGeeks”. GeeksforGeeks. Consult. 2024-06-23. [Em linha]. Disponível: https://www.geeksforgeeks.org/minimum-steps-to-reach-the-target/

[7] J. R. Bitner e E. M. Reingold, “Backtrack programming techniques”, Commun. ACM, vol. 18, n.º 11, pp. 651–656, novembro de 1975. Consult. 2024-06-23. [Em linha]. Disponível: https://doi.org/10.1145/361219.361224

[8] F. R. (Editor), P. v. B. (Editor) e T. W. (Editor), Eds., Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Sci., 2006.

[9] F. L. Bauer e H. Wössner, “The “Plankalkül” of Konrad Zuse”, Commun. ACM, vol. 15, n.º 7, pp. 678–685, julho de 1972. Consult. 2024-06-23. [Em linha]. Disponível: https://doi.org/10.1145/361454.361515