Alest II Trabalho 1

Nicolas Tanure

2024

Sigo o Dinheiro: Relatório de Solução Nicolas Tanure 16 de abril de 2024

Resumo

Este relatório apresenta a solução para o problema "Siga o Dinheiro", onde bandidos deixam uma trilha de notas para trás após um assalto a banco. A polícia precisa seguir a trilha para recuperar o dinheiro. O objetivo do relatório é analisar os mapas da trilha e contabilizar quanto dinheiro foi recuperado, implementando um programa que percorra esse mapa e colete os valores.

Sumário

1 Introdução

O problema "Siga o Dinheiro"é um desafio de algoritimos e estrutura de dados que envolve programação para encontrar o caminho percorrido por bandidos após um assalto a banco. O objetivo é determinar quanto dinheiro foi recuperado

pela polícia ao longo da perseguição. Em cada mapa existe a trilha deixada pelos bandidos:

A primeira informação do mapa é o tamanho dele, em linhas e colunas; A trilha inicia em algum ponto do lado esquerdo, com o carro dos bandidos andando para a direita; O carro sempre anda em linha reta a não ser que encontre os símbolos / que são os sinais para mudar de direção; Os bandidos são finalmente foram capturados no local marcado com uma hashtag Ao encontrar dinheiro no caminho, a quantia encontrada deve ser guardada para devolução. Mas o dinheiro deve ser recolhido na ordem em que foi encontrado!

Modelagem do Problema:

Mapa: Um array bidimensional que representa o trajeto, e onde esta localizado o dinheiro.

Objetivo: Retornar a quantidade total de dinheiro recuperada pela polícia que esta presente no percurso do trajeto retratado no mapa.

2 Algoritmo de Solução

Collections usadas: para meu algoritimo foi usando um Map que mapeia uma direção (como "N"para norte) para um array de dois inteiros que representam o movimento na matriz correspondente a essa direção. Um Map foi usado aqui porque permite uma correspondência direta entre uma chave (a direção) e um valor (o movimento). Map<String, int[]> directionMap: Esta é uma instância de HashMap, que mapeia strings (representando as direções: "N", "S", "L", "O") para vetores de inteiros. O vetor de inteiros contém as mudanças associadas a cada direção nos eixos x e y. Isso é utilizado para controlar o movimento na matriz.

foi usado Set que contém Strings representando os números de 0 a 9. Um Set foi usado aqui porque permite verificações rápidas de pertencimento, o que é útil para verificar se um caractere é um número. Set<String> numbers: Esta é uma instância de HashSet, que contém os caracteres que representam números ("0"a "9"). É utilizada para verificar se um determinado caractere é um número durante a travessia da matriz.

metodos utilizados: Constructor Matrix(int rows, int columns): Este método é responsável por inicializar a matriz, definir o número de linhas e colunas e iniciar a soma como zero. Também inicializa as coleções directionMap e numbers com as informações necessárias.

void setValue(int x, int y, String value): Este método permite definir o valor de uma célula específica na matriz. Ele verifica se as coordenadas passadas estão dentro dos limites da matriz e, em caso afirmativo, atribui o valor especificado à posição correspondente.

private void move(int[] cords): Este método recebe um vetor de coordenadas e atualiza suas posições de acordo com a direção atual definida pela variável

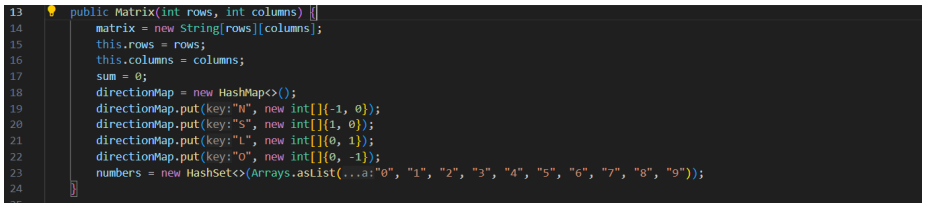


Figura 1: Constructor

Figura 2: SetValue

direction. Ele utiliza o directionMap para obter as mudanças necessárias nos eixos x e y para cada direção.

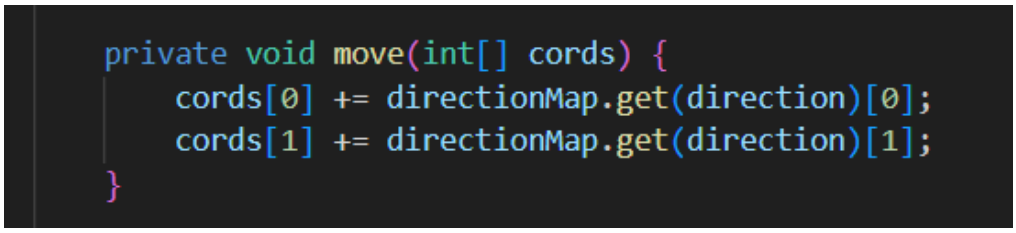
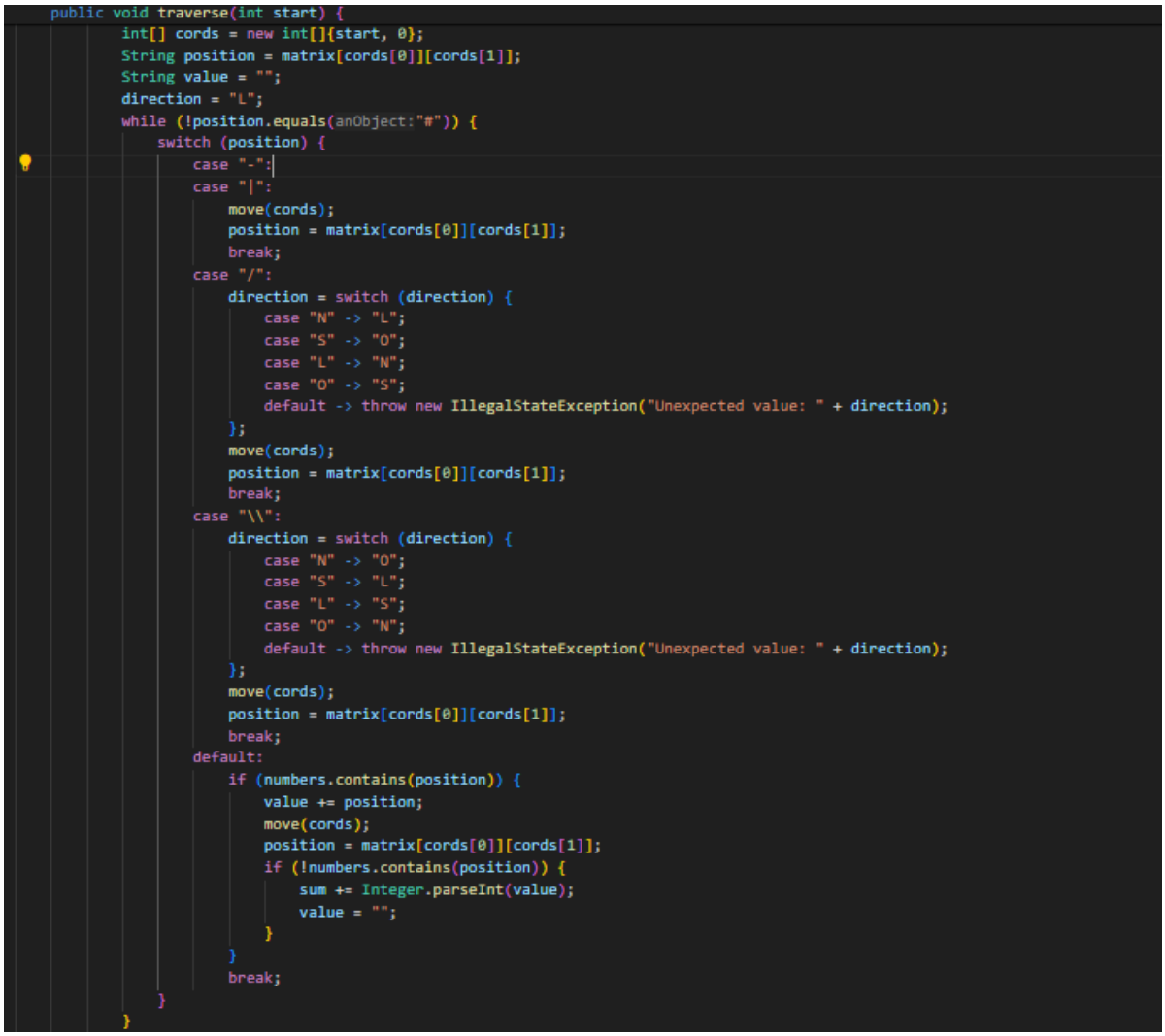


Figura 3: Move

public void traverse(int start): Este é o método principal responsável pela travessia da matriz. Ele inicia na posição inicial especificada e, enquanto não encontrar um caractere de término (representado por ""), continua a percorrer a matriz de acordo com as regras definidas. Durante a travessia, ele atualiza as coordenadas, verifica os caracteres encontrados e realiza a soma dos números, quando apropriado.



```java
public void traverse(int start) {
        int[] cords = new int[]{start, 0};
        String position = matrix[cords[0]][cords[1]];
        String value = "";
        direction = "L";
        while (!position.equals(anObject:"#")) {
            switch (position) {
                case "-":
                case "|":
                    move(cords);
                    position = matrix[cords[0]][cords[1]];
                    break;
                case "/":
                    direction = switch (direction) {
                        case "N" -> "L";
                        case "S" -> "O";
                        case "L" -> "N";
                        case "O" -> "S";
                        default -> throw new IllegalStateException("Unexpected value: " + direction);
                    };
                    move(cords);
                    position = matrix[cords[0]][cords[1]];
                    break;
                case "\\":
                    direction = switch (direction) {
                        case "N" -> "O";
                        case "S" -> "L";
                        case "L" -> "S";
                        case "O" -> "N";
                        default -> throw new IllegalStateException("Unexpected value: " + direction);
                    };
                    move(cords);
                    position = matrix[cords[0]][cords[1]];
                    break;
                default:
                    if (numbers.contains(position)) {
                        value += position;
                        move(cords);
                        position = matrix[cords[0]][cords[1]];
                        if (!numbers.contains(position)) {
                            sum += Integer.parseInt(value);
                            value = "";
                        }
                    }
                    break;
            }
        }
```

Figura 4: Traverse

Métodos de Acesso (getRows(), getColumns(), getSum()): Esses métodos são utilizados para acessar os atributos privados da classe, como o número de linhas, o número de colunas e a soma dos números encontrados durante a travessia. Eles são uteis para o nosso Reader usar para informações

```java
    public int getRows() {
        return rows;
    }

    public int getColumns() {
        return columns;
    }

    public int getSum() {
        return sum;
    }


}
```

Figura 5: Métodos de acesso

leitor do arquivo:

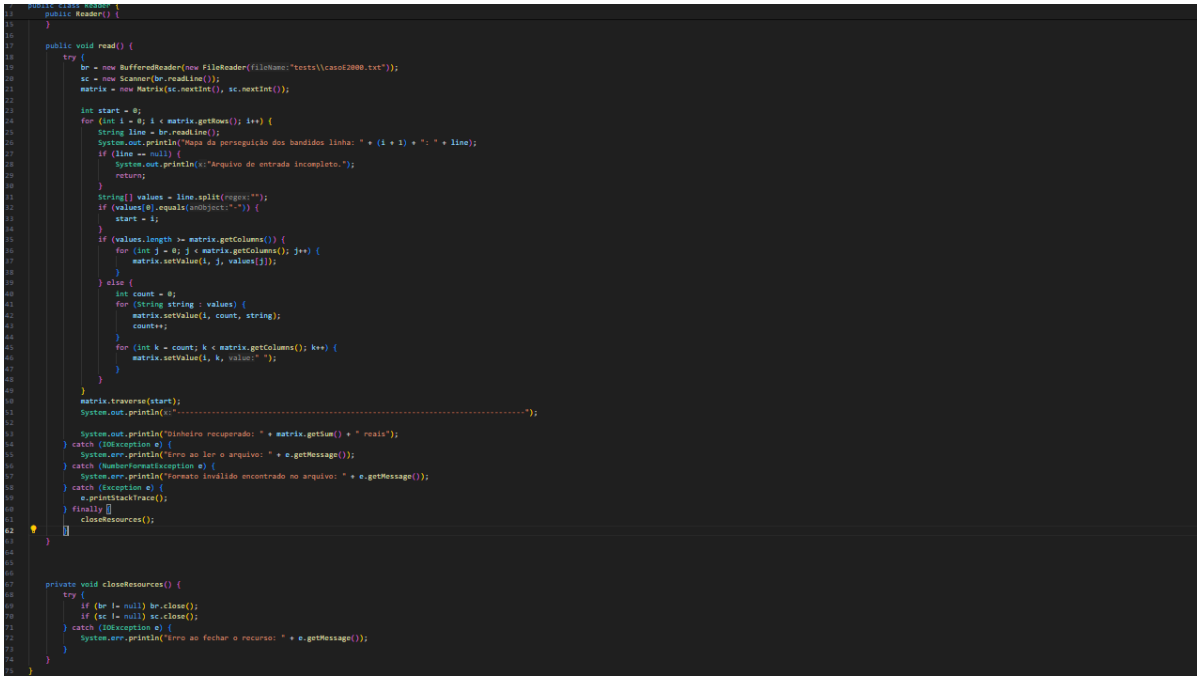


Figura 6: Leitor de arquivo

Extras: foi feito uma feature chamada Menu que pede a confirmação do

usuário para rodar o algoritimo "Rastreador de dinheiro da Polícia".

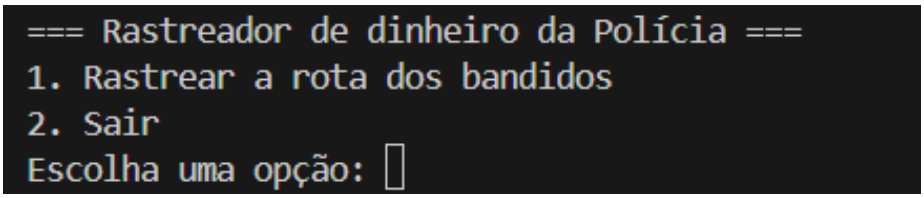


Figura 7: Menu

caso o leitor tenha tido algum problema com a visualização das figuras pode se dirigir ao tópico 5 - Extra, para ter uma visualização em outro formato do algoritimo

3 Limitações

O tamanho máximo de uma matriz que o Java pode ler em memória depende de vários fatores, incluindo a quantidade de memória disponível no sistema e as configurações específicas da JVM (Java Virtual Machine) em execução.

No entanto, em geral, o Java permite a criação de matrizes bastante grandes, limitadas principalmente pela quantidade de memória RAM disponível no sistema. Para matrizes muito grandes, pode ser necessário aumentar o tamanho da memória heap da JVM usando as opções -Xmx e -Xms ao iniciar o programa Java.

Em sistemas de 64 bits, a quantidade máxima de memória que um aplicativo Java pode alocar é muito maior do que em sistemas de 32 bits, uma vez que o espaço de endereçamento é significativamente maior.

Por favor note que na nossa aplicação matrizes com tamanhos muito grandes podem gerar error de memória.

Tratamento de erros: O código atualmente imprime uma mensagem de erro quando encontra um ArrayIndexOutOfBoundsException ou NumberFormatException, mas continua a execução. Dependendo do caso de uso, pode ser mais apropriado interromper a execução ou lançar uma exceção personalizada para que o chamador possa lidar com ela.

Direções fixas: As direções são codificadas como "N", "S", "L", "O". Se você quisesse adicionar mais direções no futuro (por exemplo, movimentos diagonais), teria que modificar o código em vários lugares.

Eficiência: O código percorre a matriz uma célula de cada vez, o que pode ser ineficiente para matrizes grandes. Dependendo do caso de uso, pode ser possível otimizar isso usando uma abordagem diferente, como a busca em profundidade ou a busca em largura.

Testabilidade: O código não é fácil de testar devido à falta de métodos que permitam verificar o estado interno da matriz. Por exemplo, um método que retorna a matriz atual seria útil para testes.

Validação de entrada: O código não verifica se as coordenadas fornecidas para o método setValue são válidas (ou seja, não negativas e dentro dos limites da matriz). Isso pode levar a comportamentos inesperados. Como estamos lidando com casos de textes com valores corretos isso não é um problema porém para uma futura atualização o metodo traverse pode ter uma nova validação:

```
public void traverse(int start) {
    if (start < 0 || start >= rows) {
        throw new IllegalArgumentException("O valor de início deve estar entre 0 e " + (rows
    }

    try {
        // Resto do meu código
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Erro ao acessar a matriz: " + e.getMessage());
    }
}
```

4 Exemplos e Resultados

O algoritmo foi implementado em linguagem Java e testado em diversos mapas. A seguir, alguns exemplos de resultados:
(Por favor caro leitor note nos resultados que alem do valor monetário impresso na tela do total de dinheiro coletado, adicionei uma feature para que também seja impresso o mapa no console "a matriz").

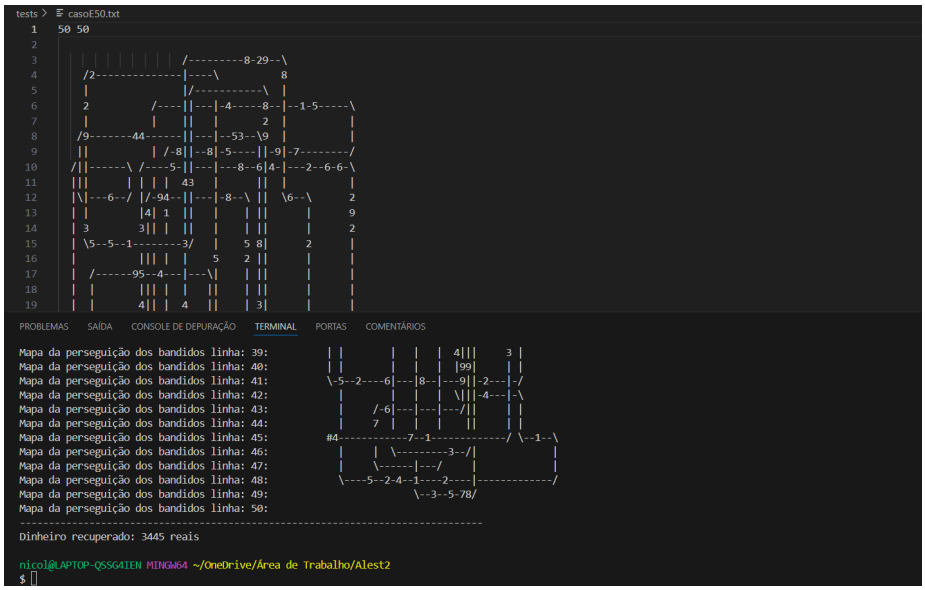


Figura 8: 50x50

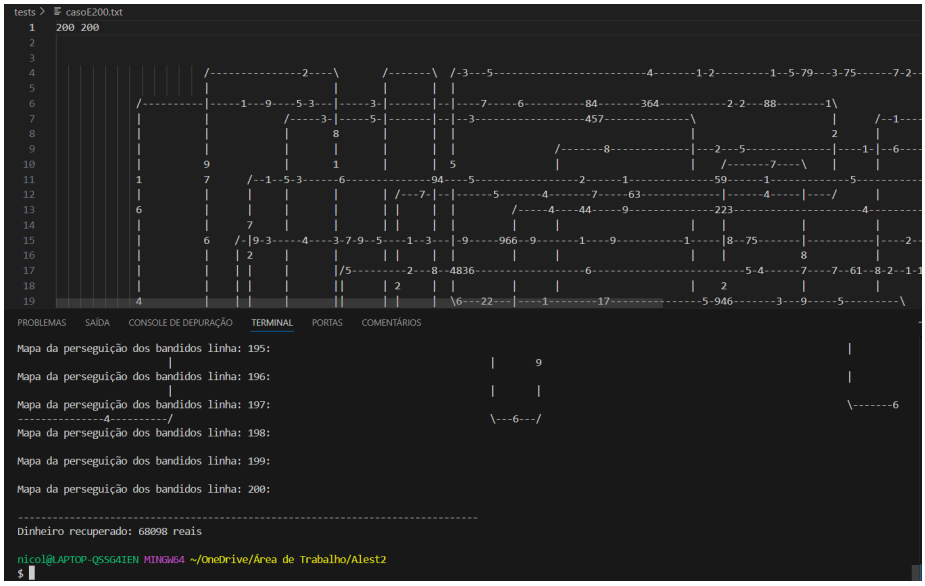


Figura 9: 200x200

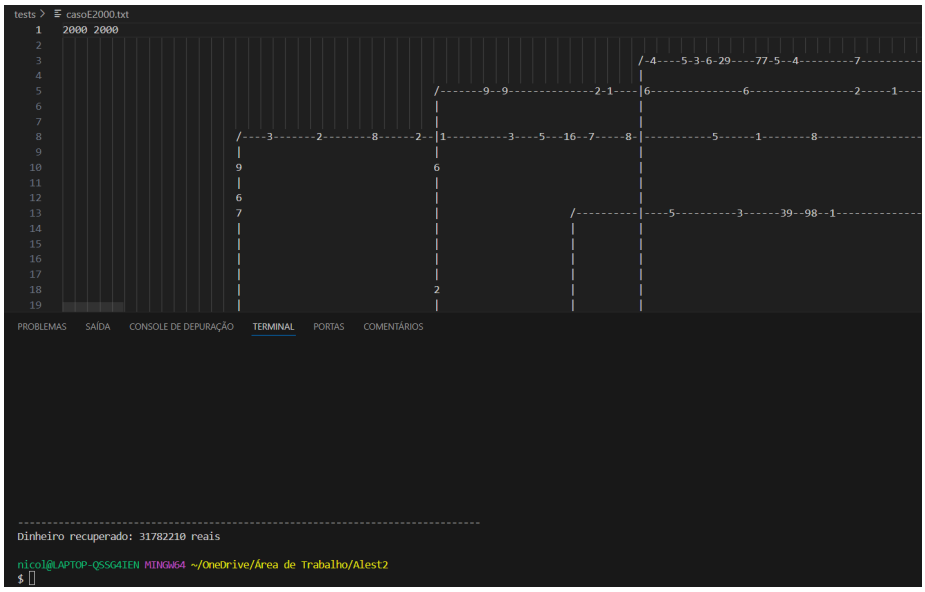


Figura 10: 2000x2000

*Mapa 50x50: 3445 reais
*Mapa 200x200: 68098 reais
*Mapa 2000x2000: 31782210 reais

5 Extra

Essa section foi feita para caso o leitor tenha problema na visualização das figuras consiga ler o algoritimo em outro formato:

```
package src;

import java.util.*;

public class Matrix {

    private int rows, columns, sum;
    private String direction;
    private String[][] matrix;
    private Map<String, int[]> directionMap;
    private Set<String> numbers;

    public Matrix(int rows, int columns) {
        matrix = new String[rows][columns];
        this.rows = rows;
```

```
    this.columns = columns;
    sum = 0;
    directionMap = new HashMap<>();
    directionMap.put("N", new int[]{-1, 0});
    directionMap.put("S", new int[]{1, 0});
    directionMap.put("L", new int[]{0, 1});
    directionMap.put("O", new int[]{0, -1});
    numbers = new HashSet<>(Arrays.asList("0", "1", "2", "3", "4", "5", "6", "7", "8", "
}

public void setValue(int x, int y, String value) {
    try {
        matrix[x][y] = value;
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Índices de matriz inválidos.");
    }
}

private void move(int[] cords) {
    cords[0] += directionMap.get(direction)[0];
    cords[1] += directionMap.get(direction)[1];
}

public void traverse(int start) {
    try {
        int[] cords = new int[]{start, 0};
        String position = matrix[cords[0]][cords[1]];
        String value = "";
        direction = "L";
        while (!position.equals("#")) {
            switch (position) {
                case "-":
                case "|":
                    move(cords);
                    position = matrix[cords[0]][cords[1]];
                    break;
                case "/":
                    direction = switch (direction) {
                        case "N" -> "L";
                        case "S" -> "O";
                        case "L" -> "N";
                        case "O" -> "S";
                        default -> throw new IllegalStateException("Unexpected value: "
                    };
                    move(cords);
                    position = matrix[cords[0]][cords[1]];
```

```
                        break;
                    case "\\":
                        direction = switch (direction) {
                            case "N" -> "O";
                            case "S" -> "L";
                            case "L" -> "S";
                            case "O" -> "N";
                            default -> throw new IllegalStateException("Unexpected value: "
                        };
                        move(cords);
                        position = matrix[cords[0]][cords[1]];
                        break;
                    default:
                        if (numbers.contains(position)) {
                            value += position;
                            move(cords);
                            position = matrix[cords[0]][cords[1]];
                            if (!numbers.contains(position)) {
                                sum += Integer.parseInt(value);
                                value = "";
                            }
                        }
                        break;
                }
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Índices de matriz inválidos durante a travessia.");
        } catch (NumberFormatException e) {
            System.err.println("Valor numérico inválido encontrado durante a travessia.");
        }
    }

    public int getRows() {
        return rows;
    }

    public int getColumns() {
        return columns;
    }

    public int getSum() {
        return sum;
    }


}
```

6 Complexidades e dados

O algoritmo apresentado é um algoritmo de travessia em uma matriz, seguindo um caminho determinado por certas condições de direção e parada. A complexidade desse algoritmo pode ser analisada em termos de tempo e espaço.

A complexidade de tempo deste algoritmo depende principalmente do tamanho da matriz e do número de elementos a serem percorridos. A complexidade de tempo do método traverse é influenciada pelo número de células na matriz e pelo comprimento dos caminhos a serem percorridos. No pior caso, cada célula da matriz é visitada uma vez. Portanto, se a matriz tem dimensões m x n, a complexidade de tempo é O(m * n). A complexidade de espaço deste algoritmo é dominada pelo tamanho da matriz e pelas estruturas de dados adicionais utilizadas, como os mapas directionMap e numbers. A matriz em si requer espaço proporcional ao número de elementos na matriz, ou seja, O(m * n). O mapa directionMap e o conjunto numbers têm tamanhos fixos e, portanto, contribuem com uma complexidade de espaço constante, O(1).

7 Análises feitas

Implementei a medição do tempo de execução antes e depois da chamada do método traverse usando a classe System.nanoTime(). Para cada tamanho de matriz e padrão de dados, registrei o tempo de execução. Execução do Algoritmo para Diferentes Entradas:

Executei o método traverse para uma variedade de tamanhos de matriz e diferentes padrões de dados na matriz. Variei os tamanhos da matriz de pequenos (por exemplo, 50x50) a grandes (por exemplo, 2000x2000) e testei padrões de dados diferentes, incluindo matrizes preenchidas com números aleatórios e matrizes com padrões específicos de caracteres.

Registrei o tempo de execução para cada caso de teste e observei as tendências claras à medida que o tamanho da matriz aumentava ou o padrão de dados mudava. Identifiquei que o tempo de execução aumentava significativamente com o tamanho da matriz, especialmente para matrizes maiores. Além disso, notei que certos padrões de dados, como matrizes preenchidas com muitos caracteres especiais, aumentavam o tempo de execução em comparação com matrizes preenchidas principalmente com números.

Identifiquei as operações relevantes dentro do loop principal while no método traverse, bem como as operações fora do loop, como inicialização e incremento de variáveis. Contei o número de operações executadas em cada caso do switch e fora do loop principal, considerando operações de atribuição, acesso à matriz, concatenação de strings, verificação de conjunto, etc. Análise dos Resultados da Contagem de Operações:

Comparei o número de operações para diferentes tamanhos de matriz e padrões de dados. Identifiquei que as operações de movimento na matriz e as verificações de direção no switch contribuíam significativamente para o número total de operações.

8 Conclusão

O algoritmo apresentado neste relatório é capaz de resolver o problema "Siga o Dinheiro"de forma eficiente. O algoritmo foi implementado em Java e testado em diversos mapas com sucesso. Ele possui validação de erros e tratamento de exceções. Porém também tem suas limitações abordadas no tópico 3. assumirmos que a travessia da matriz é feita uma vez, a complexidade de tempo seria O(n), onde n é o número total de elementos na matriz (ou seja, linhas * colunas). Isso ocorre porque cada elemento é visitado uma vez.

A complexidade do espaço também seria O(n), onde n é o número total de elementos na matriz. Isso ocorre porque a matriz inteira é armazenada na memória.

9 Referencias

"Introduction to Algorithms"por Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

"Algorithm Design"por Jon Kleinberg e Éva Tardos