Lost

André Santos, nº55415
Tiago Teles, nº54953

Variáveis

aK = Arcos do grafo da Kate (do tipo: ({W,G,n}, {W,G,n,X}), onde n identifica uma roda mágica e os conjuntos indicam os tipos de casa atribuíveis à tail e à head do arco respetivamente)

vK = Vértices com arcos incidentes do grafo da Kate (todos os aK)

aJ = Arcos do grafo do John (do tipo: ({G,n,m}, {G,n,m,X}), onde n identifica uma roda mágica, m o destino de uma roda mágica, e os conjuntos indicam os tipos de casa atribuíveis à tail e à head do arco respetivamente)

vJ = Vértices com arcos incidentes do grafo do John (todos os aJ) (trata-se de um valor com um excesso máximo de M devido à forma como é obtido durante a criação do grafo)

Complexidade Temporal

Criar grafo de listas de sucessores: Θ(L)

Criar grafos para John e Kate: Θ(R*C) (as operações internas são todas Θ(1))

Bellman Ford em lista de arcos para o John: O(vJ*aJ)

Dijkstra em lista de sucessores com fila de prioridades para a Kate: O((vK+aK) * log vK)

Complexidade total (N casos): O(N*((R*C) + (vJ*aJ) + ((vK+aK) * log vK)))

Complexidade Espacial

Grelha da ilha: Θ(R*C)

Vetor rodas mágicas: Θ(M)

Grafo para a Kate (listas de sucessores): Θ(R*C + aK)

Grafo para o John (lista de arcos): Θ(aJ)

Matriz de nós processados Kate: Θ(R*C)

Matriz de comprimentos mínimos de nós Kate: Θ(R*C)

Fila com prioridades Kate: Θ(vK)

Matriz de comprimentos mínimos de nós John: Θ(R*C)

Complexidade total: Θ(R*C + aK), visto que aK > aJ dado que existe sempre um mínimo de casas com água maior do que o máximo de casas mágicas para o grafo em questão e que vK < R*C visto que os vértices que contribuem a vK formam um subconjunto da grelha.

Conclusões

Em termos de complexidade temporal a nossa solução pode ainda ser melhorada no que toca à componente originária do cálculo do comprimento do menor caminho referente à Kate (O((vK+aK) * log vK)). Isto porque os pesos dos arcos relevantes à Kate têm todos como peso 1 ou 2, o que, com algumas modificações, possibilita um algoritmo análogo a uma pesquisa por largura que teria como complexidade temporal limite O(vK + 2aK) -> O(vK + aK).

Vemos duas formas de alcançar este resultado:

A primeira, e talvez a mais simples, requer apenas que, ao criar o grafo, se desdobrem as ligações com peso 2 em duas de peso 1 recorrendo a um nó auxiliar, sendo este grafo processável pela pesquisa em largura normal. Esta solução tem como desvantagem o aumento do tamanho absoluto do grafo.

A segunda passa por implementar Dijkstra com recurso, não a uma fila de prioridades assente num heap, mas sim a um vetor circular de sacos (com dimensão 2 neste caso, um para cada peso). Esta estrutura poderia ser implementada num Set de nós de listas duplamente ligadas que implementariam os tais sacos (à semelhança de LinkedHashSet mas utilizando um Set comum para ambas as listas, visto que a informação contida nas mesmas é sempre disjunta). Desta forma o algoritmo seria idêntico ao que acabámos por implementar com a diferença de que as operações a realizar sobre a fila teriam complexidade Θ(1). Esta solução tem como desvantagem o uso de estruturas de dados auxiliares de dimensões acrescidas. Acabámos também por descobrir mais tarde que isto se trata de uma adaptação do algoritmo de Dial.

De entre estas duas a mais indicada seria a primeira no caso de se pretender que este fosse o único uso do grafo e a segunda caso contrário.

Em relação às restantes componentes, Θ(R*C) para a iteração da grelha não pode ser diminuída pois, com exceção da fronteira exterior, não dispomos de conhecimento prévio em relação a cada posição da grelha e O(vJ*aJ) para a execução de Bellman Ford talvez pudesse ser reduzido graças ao conhecimento acerca da topologia do grafo, mas não vemos de forma imediata como.

Em termos de complexidade espacial, Θ(R*C) originário da grelha da ilha pode ser reduzido para Θ(C) com o uso de uma matriz circular pois no processamento de cada vértice são necessários apenas os que lhe são adjacentes à esquerda e acima. Alternativamente, sacrificando a otimização anterior seria possível diminuir Θ(R*C + aK) do grafo da Kate para Θ(vK + aK) fazendo uma primeira iteração da ilha para contabilizar os vértices válidos e realizando de seguida uma reindexação dos mesmos. Ainda outra possibilidade seria fazer uso de um LinkedHashMap, que permitiria ter ambas as reduções em simultâneo a troco de se ter uma estrutura esparsa. Poderiam também ser feitas alterações semelhantes às estruturas de auxílio à execução de Dijkstra para reduzir o seu gasto de memória de Θ(R*C) para Θ(vK). O mesmo se aplica para as estruturas do Bellman Ford mas com a complexidade a passar a Θ(vJ) (ou menos, visto tornar-se possível eliminar o excesso de vJ com a iteração adicional da ilha e recolha dos destinos das rodas mágicas). A fila com prioridades utilizada para o Dijkstra também poderia ter dimensões inferiores ao número de vértices do grafo visto que temos algum conhecimento em relação à topologia do grafo (disposição em grelha). Esta seria uma alteração trivial mas não realizámos os cálculos necessários para saber qual seria a dimensão ótima ou aproximada desta.

Já em termos absolutos espaciais seria possível armazenar as listas de sucessores em listas ligadas simples em vez de listas duplamente ligadas, visto que estas não nos trazem mais valias (ou até mesmo vetores, analisando todas as casas adjacentes aquando da iteração da grelha para se ou alocando vetores de 4 posições, o que, apesar de ser um ligeiro excesso, provavelmente não se revelaria problemático). Provavelmente teria sido possível utilizar uma estrutura híbrida em vez de dois grafos completamente independentes para a Kate e o John sem sacrificar as complexidades temporais, mas a implementação da mesma traduzir-se-ia em complexidade adicional do código e potencialmente uma redução dos tempos absolutos pelo que decidimos não o fazer. Uma possível implementação seria utilizar um grafo em vetor de pares de listas de sucessores, uma para água e outra para relva acompanhado de uma lista ligada dos arcos com sucessores válidos para o John. Ter-se-ia ainda uma lista de arcos para as rodas mágicas.

Na vertente temporal, ainda em termos absolutos, poderíamos ter tornado a iteração da grelha ligeiramente mais eficiente programando por extenso os casos limite (início de linha e encontrar um arco pertencente ao grafo da Kate e do John) mas a quantidade de código adicional não se justificava.

```java
import java.io.IOException;

public class Main {

    public static void main(String[] args) throws IOException {
        try (TurboScanner in = new TurboScanner(System.in)) {
            byte N = (byte) in.nextInt();
            for (byte testCase = 1; testCase <= N; testCase++) {
                byte R = (byte) in.nextInt();
                byte C = (byte) in.nextInt();
                byte M = (byte) in.nextInt();

                byte[][] grid = new byte[R][C];
                LostBuilder builder = new LostBuilder(grid, R, C, M);

                for (byte j = 0; j < C; j++) {
                    grid[0][j] = in.nextByte();
                }
                in.nextByte();
                for (byte i = 0; i < R - 1; i++) {
                    for (byte j = 0; j < C; j++) {
                        grid[i + 1][j] = in.nextByte();
                        builder.processCell(i, j);
                    }
                    in.nextByte();
                }
                for (byte j = 0; j < C; j++) {
                    builder.processCell((byte) (R - 1), j);
                }

                for (int i = 0; i < M; i++) {
                    byte r = (byte) in.nextInt();
                    byte c = (byte) in.nextInt();
                    int t = in.nextInt();
                    builder.addMagicalEdge(i, r, c, t);
                }
                byte rj = (byte) in.nextInt();
                byte cj = (byte) in.nextInt();
                byte rk = (byte) in.nextInt();
                byte ck = (byte) in.nextInt();
                builder.setJohnStart(rj, cj);
                builder.setKateStart(rk, ck);

                System.out.println("Case #" + testCase);
                System.out.println(builder.build().result());
            }
        }
    }
}

public class Edge {

    final byte iTail;
    final byte jTail;
    final byte iHead;
    final byte jHead;
    final int weight;

    public Edge(byte iTail, byte jTail, byte iHead, byte jHead, int weight) {
        this.iTail = iTail;
        this.jTail = jTail;
        this.iHead = iHead;
        this.jHead = jHead;
        this.weight = weight;
    }
}
```

```java
import java.util.LinkedList;
import java.util.List;

public class LostBuilder {

    private static final byte
            W = (byte) 'W',
            G = (byte) 'G',
            O = (byte) 'O',
            X = (byte) 'X';

    private enum Directions {
        TOP((byte) -1, (byte) 0),
        LEFT((byte) 0, (byte) -1);

        byte vertical, horizontal;

        Directions(byte vertical, byte horizontal) {
            this.vertical = vertical;
            this.horizontal = horizontal;
        }
    }

    private final byte[][] grid;
    private final List<WeightedSuccessor>[][] kateGraph;
    private final List<Edge> johnGraph;
    private final byte[] johnStart;
    private final byte[] kateStart;
    private final byte[] magicalCells;
    private final byte[] exit;
    private short kateVertices, johnVertices;

    @SuppressWarnings("unchecked")
    public LostBuilder(byte[][] grid, byte rows, byte columns, byte numMagicalCells) {
        this.grid = grid;
        this.johnGraph = new LinkedList<>();
        this.kateGraph = (List<WeightedSuccessor>[][]) new List[rows][columns];
        this.johnStart = new byte[2];
        this.kateStart = new byte[2];
        this.magicalCells = new byte[2 * numMagicalCells];
        this.exit = new byte[2];
        this.kateVertices = 0;
        this.johnVertices = 0;
    }

    public Lost build() {
        // Add the exit (+1) and the potentially not accounted for, but accessible via magical whe-
els, islands
        short johnVertices = (short) (this.johnVertices + 1 + magicalCells.length / 2);
        return new Lost(kateGraph, johnGraph, kateStart, johnStart, exit, kateVertices, johnVerti-
ces);
    }

    public void processCell(byte i, byte j) {
        byte cell = grid[i][j];
        boolean outgoingEdge = true;
        switch (cell) {
            case X:
                exit[0] = i;
                exit[1] = j;
                outgoingEdge = false;
```

```
        case G:
            //john
            boolean added = false;
            for (Directions dir : Directions.values()) {
                byte i1 = (byte) (i + dir.vertical);
                byte j1 = (byte) (j + dir.horizontal);
                if (inBounds(i1, j1) && johnCanMoveTo(grid[i1][j1])) {
                    if (outgoingEdge) {
                        johnGraph.add(new Edge(i, j, i1, j1, weightFrom(cell)));
                        added = true;
                    }
                    if (grid[i1][j1] != X) {
                        johnGraph.add(new Edge(i1, j1, i, j, weightFrom(grid[i1][j1])));
                    }
                }
            }
            if (added) {
                johnVertices++;
            }
        case W:
            //kate
            for (Directions dir : Directions.values()) {
                byte i1 = (byte) (i + dir.vertical);
                byte j1 = (byte) (j + dir.horizontal);
                if (inBounds(i1, j1) && kateCanMoveTo(grid[i1][j1])) {
                    if (outgoingEdge) {
                        List<WeightedSuccessor> origin = kateGraph[i][j];
                        if (origin == null) {
                            kateVertices++;
                            kateGraph[i][j] = origin = new LinkedList<>();
                        }
                        origin.add(new WeightedSuccessor(i1, j1, weightFrom(cell)));
                    }
                    List<WeightedSuccessor> destiny = kateGraph[i1][j1];
                    if (destiny == null) {
                        kateVertices++;
                        kateGraph[i1][j1] = destiny = new LinkedList<>();
                    }
                    if (grid[i1][j1] != X) {
                        destiny.add(new WeightedSuccessor(i, j, weightFrom(grid[i1][j1])));
                    }
                }
            }
            break;
        case O:
            break;
        default:
            assert ((byte) '1' <= cell && cell <= (byte) '9');
            magicalCells[2 * (cell - '1')] = i;
            magicalCells[2 * (cell - '1') + 1] = j;
            grid[i][j] = G;
            processCell(i, j);
            break;
    }
}

public void addMagicalEdge(int magicalCell, byte i, byte j, int weight) {
    johnGraph.add(new Edge(magicalCells[2 * magicalCell], magicalCells[2 * magicalCell + 1], i, j,
weight));
}

public void setJohnStart(byte i, byte j) {
    johnStart[0] = i;
    johnStart[1] = j;
}
```

```
    public void setKateStart(byte i, byte j) {
        kateStart[0] = i;
        kateStart[1] = j;
    }

    private boolean kateCanMoveTo(byte cell) {
        return cell != O;
    }

    private boolean johnCanMoveTo(byte cell) {
        switch (cell) {
            case X:
            case G:
            default:
                return true;
            case W:
            case O:
                return false;
        }
    }

    private byte weightFrom(byte cell) {
        switch (cell) {
            case X:
            case G:
            default:
                return 1;
            case W:
                return 2;
            case O:
                return -1;
        }
    }

    private boolean inBounds(int i, int j) {
        return i < grid.length && i >= 0 &&
                j < grid[0].length && j >= 0;
    }
}

import java.util.*;

public class Lost {

    private final List<WeightedSuccessor>[][] kateGraph;
    private final List<Edge> johnGraph;
    private final byte[] johnStart;
    private final byte[] kateStart;
    private final byte[] exit;
    private final short kateVertices;
    private final short johnVertices;

    public Lost(List<WeightedSuccessor>[][] kateGraph, List<Edge> johnGraph, byte[] kateStart,
byte[] johnStart, byte[] exit, short kateVertices, short johnVertices) {
        this.kateGraph = kateGraph;
        this.johnGraph = johnGraph;
        this.johnStart = johnStart;
        this.kateStart = kateStart;
        this.exit = exit;
        this.kateVertices = kateVertices;
        this.johnVertices = johnVertices;
    }

    public String result() {
        return calcJohn() + "\n" + calcKate();
    }
```

```java
public String calcKate() {
    if (Arrays.equals(kateStart, exit)) {
        return "Kate 0";
    }
    int kateLength = dijkstra();
    return (kateLength != Integer.MAX_VALUE) ? "Kate " + kateLength : "Kate Unreachable";
}

private int dijkstra() {
    int[][] length = new int[kateGraph.length][kateGraph[0].length];
    boolean[][] selected = new boolean[kateGraph.length][kateGraph[0].length];
    PriorityQueue<WeightedSuccessor> connected = new PriorityQueue<>(kateVertices, Comparator.com-
paringInt(o -> o.weight));
    for (int[] ints : length) {
        Arrays.fill(ints, Integer.MAX_VALUE);
    }
    connected.add(new WeightedSuccessor(kateStart[0], kateStart[1], (byte) 0));
    length[kateStart[0]][kateStart[1]] = 0;
    do {
        WeightedSuccessor node = connected.poll();
        if (node.i == exit[0] && node.j == exit[1]) {
            break;
        }
        selected[node.i][node.j] = true;
        if (kateGraph[node.i][node.j] != null) {
            for (WeightedSuccessor v : kateGraph[node.i][node.j]) {
                if (!selected[v.i][v.j]) {
                    int newLength = length[node.i][node.j] + v.weight;
                    if (newLength < length[v.i][v.j]) {
                        length[v.i][v.j] = newLength;
                        boolean nodeIsInQueue = length[v.i][v.j] < Integer.MAX_VALUE;
                        if (nodeIsInQueue) {
                            connected.remove(new WeightedSuccessor(v.i, v.j, 0));
                        }
                        connected.add(new WeightedSuccessor(v.i, v.j, length[v.i][v.j]));
                    }
                }
            }
        }
    } while (!connected.isEmpty());
    return length[exit[0]][exit[1]];
}

public String calcJohn() {
    if (Arrays.equals(johnStart, exit)) {
        return "John 0";
    }
    int[][] length = new int[kateGraph.length][kateGraph[0].length];
    for (int[] ints : length)
        Arrays.fill(ints, Integer.MAX_VALUE);

    length[johnStart[0]][johnStart[1]] = 0;
    boolean changes = true;
    for (int i = 1; i < johnVertices && changes; i++) {
        changes = updateLengths(length);
    }
    if (changes && updateLengths(length))
        return "John Lost in Time";

    int distance = length[exit[0]][exit[1]];
    return (distance == Integer.MAX_VALUE) ? "John Unreachable" : "John " + distance;
}
```

```
    boolean updateLengths(int[][] len) {
        boolean changes = false;
        for (Edge edge : johnGraph) {
            int iTail = edge.iTail;
            int jTail = edge.jTail;
            int iHead = edge.iHead;
            int jHead = edge.jHead;
            if (len[iTail][jTail] < Integer.MAX_VALUE) {
                int newLen = len[iTail][jTail] + edge.weight;
                if (newLen < len[iHead][jHead]) {
                    len[iHead][jHead] = newLen;
                    changes = true;
                }
            }
        }
        return changes;
    }

}


public class WeightedSuccessor {

    final byte i;
    final byte j;
    final int weight;

    public WeightedSuccessor(byte i, byte j, int weight) {
        this.i = i;
        this.j = j;
        this.weight = weight;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        WeightedSuccessor that = (WeightedSuccessor) o;

        if (i != that.i) return false;
        return j == that.j;
    }

    @Override
    public int hashCode() {
        int result = i;
        result = 31 * result + (int) j;
        return result;
    }
}
```

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

class TurboScanner implements AutoCloseable{
    BufferedReader br;

    public TurboScanner(InputStream in) {
        br = new BufferedReader(new InputStreamReader(in));
    }

    public int nextInt() throws IOException {
        return Integer.parseInt(next());
    }

    public String next() throws IOException {
        int i = 0;
        char ascii = '_';
        char[] chars = new char[13];
        while (ascii != ' ' && ascii != '\n') {
            ascii = (char) br.read();
            chars[i++] = ascii;
        }
        return new String(chars).trim();
    }

    public String nextLine() throws IOException {
        return br.readLine();
    }

    public byte nextByte() throws IOException {
        //Will break with non ascii characters, be aware
        return (byte) br.read();
    }
    public void close() throws IOException {
        br.close();
    }
}
```