Algorisme de Backtracking per al KenKen

1 Introducció

En aquest document, presentem l'algorisme de backtracking utilitzat per resoldre KenKen, un puzle que combina elements de Sudoku amb operacions aritmètiques dins de celdes agrupades. L'algorisme està implementat mitjançant dues classes principals: `Validator` i `Generator`. La classe `Validator` és responsable de validar un tauler de KenKen, mentre que la classe `Generator` s'encarrega de generar taulers de KenKen inicials.

2 Classes Principals

A continuació, es descriuen les classes `Cell`, `Region` i `InfoCoord`, que són fonamentals per a la representació i manipulació del tauler de KenKen.

2.1 Classe Cell

La classe `Cell` representa una cel·la en una quadrícula amb coordenades. Proporciona mètodes per recuperar i comparar coordenades de les cel·les i per introduir un nombre en la cel·la.

Atributs Principals:

- `int first`: coordenada de la fila.
- `int second`: coordenada de la columna.

Mètodes Principals:

- `Cell(int first, int second)`: constructor que inicialitza les coordenades.
- `boolean equals(Object obj)`: compara si dues cel·les són iguals.
- `int hashCode()`: retorna el codi hash de la cel·la.

La classe `Cell` permet la creació d'objectes cel·la amb coordenades específiques i implementa els mètodes equals i hashCode per assegurar la correcta comparació i ús en estructures de dades que requereixen hashing, com ara els mapes o conjunts.

2.2 Classe Region

La classe `Region` representa una regió dins del tauler KenKen, que conté detalls com l'operació aritmètica, el resultat objectiu i una llista de coordenades que identifiquen les cel·les que componen la regió.

Atributs Principals:

- `int oper`: tipus d'operació aritmètica.
- `int res`: resultat esperat de l'operació.
- `int numCoord`: nombre de cel·les en la regió.
- `List<Cell<Integer, Integer>> coordList`: llista de coordenades de les cel·les.

Mètodes Principals:

- `Region(int oper, int res, int numCoord, List<Cell<Integer, Integer>> coordList)`: constructor que inicialitza els atributs.

- `int getOper()`: retorna el tipus d'operació.
- `int getRes()`: retorna el resultat esperat.
- `int getNumCoord()`: retorna el nombre de cel·les.
- `List<Cell<Integer, Integer>> getCoordList()`: retorna la llista de coordenades.

La classe `Region` permet la creació d'objectes que encapsulen la informació necessària per a cada regió del KenKen, incloent l'operació aritmètica que s'aplica, el resultat esperat d'aquesta operació, i les cel·les que formen part de la regió.

2.3 Classe InfoCoord

La classe `InfoCoord` ajuda a emmagatzemar la informació d'una regió en el tauler, incloent la posició de la regió, l'operació aritmètica aplicada i el nombre de cel·les que comprèn la regió.

Atributs Principals:

- `Integer posReg`: posició de la regió.
- `Integer op`: operació aritmètica.
- `int nReg`: nombre de cel·les de la regió.

Mètodes Principals:

- `InfoCoord(int posReg, int op, int nReg)`: constructor que inicialitza els atributs.
- `int getPosReg()`: retorna la posició de la regió.
- `int getOp()`: retorna l'operació aritmètica.
- `int getNReg()`: retorna el nombre de cel·les.

La classe `InfoCoord` permet la creació d'objectes que emmagatzemen la informació essencial d'una regió, incloent la seva posició, l'operació aritmètica que s'aplica, i el nombre de cel·les que formen la regió.

3 Classe Validator

La classe `Validator` és responsable de validar un tauler de KenKen. Utilitza un algoritme de backtracking per assegurar-se que el tauler compleix amb totes les restriccions definides.

Atributs Principals:

- `int[][] kenken`: representació del tauler KenKen.
- `enum Operator`: enumeració dels operadors aritmètics disponibles.

Atributs Secundaris:

- `int[][] rows`: representació de les files per numero, quan hi ha un numero al tauler kenken, es ficará a la casella i = fila(on es situa al kenken), j = numero -1 que esta representat al kenken.
- `int[][] columns`: representació de les columnes per numero, quan hi ha un numero al tauler kenken, es ficará a la casella i = columna(on es situa al kenken), j = numero -1 que esta representat al kenken.
- `int[] resDefReg`: En aquesta llista es fica el resultat final de la regió.
- `int[][] resReg`: el valor temporal que te una regió, aumenta a mida que s'insereixen números al tauleri a la regió.
- `Map<Cell<Integer, Integer>, InfoCoord<Integer, Integer, Integer>> regionMap`: en aquest mapa es guarda como a key una cella i el seu valor es l'informació de la regió on es ubicada la cella.
- `int[][] region`: Es una matriu d'adjacencies on guardem a cada fila les posicions/celles de cada regio

- `int[] actNReg`: Cada casella representa una regió i el seu valor es el numero actual de casellas que no hi son dins de d'aquesta regió.
- `int IJPosReg`: representa l'index de la regio on s'ubica la i i la j.

Mètodes Principals:

- `boolean backtrackValidationSUM(...)`: verifica les restriccions per a la suma.
- `boolean backtrackValidationMULT(...)`: verifica les restriccions per a la multiplicació.
- `boolean backtrackValidationREST(...)`: verifica les restriccions per a la resta.
- `boolean backtrackValidationDIV(...)`: verifica les restriccions per a la divisió.
- `boolean backtrackValidationSUMOFSQUARES(...)`: verifica les restriccions per a la suma de nombres al cuadrad.
- `boolean backtrackValidationMOD(...)`: verifica les restriccions per a la modulitat.
- `boolean backtrackValidation(...)`: algorisme de backtracking general per validar el tauler.
- `boolean Validation(...)`: inicialitza estructures de dades i invoca el backtracking.
- `void readFile()`: llegeix un fitxer de configuració del tauler i valida el KenKen.

El mètode `backtrackValidationSUM` verifica si la suma actual més el número proposat no excedeix el resultat desitjat i comprova si es compleix la restricció quan només queda una cel·la per omplir. De manera similar, els mètodes `backtrackValidationMULT`, `backtrackValidationREST`, `backtrackValidationDIV`, `backtrackValidationSUMOFSQUARE`, `backtrackValidationMOD` fan el mateix per a les seves operacions. El mètode `backtrackValidation` implementa l'algorisme de backtracking que recorre recursivament el tauler, provant col·locar números vàlids a cada cel·la segons les restriccions.

El mètode `Validation` inicialitza les estructures de dades necessàries i invoca el mètode de backtracking per validar el tauler. Finalment, el mètode `readFile` llegeix un fitxer de configuració del tauler, i utilitza el mètode de validació per determinar si el tauler és vàlid o no.

3.1 Funcionament General de l'Algoritme de Validació

3.1.1 Inicialització i Configuració Inicial

En aquest pas es preparen totes les estructures de dades necessàries i s'estableix l'estat inicial del tauler. Es defineixen les matrius per a les restriccions de files i columnes, així com els vectors per als resultats de les regions. En aquesta part es calcula la lògica en cas de que el kenken no fos buit i contingui certs valors ja omplert per al usuari.

Listing 1: Inicialització del Validador

```
public boolean Validation(int n, int r, int[][] region, int[][] kenken) {
    int[][] rows = new int[n][n + 1];
    int[][] cols = new int[n][n + 1];
    this.kenken = kenken;
    // Inicialitzacio de les estructures de dades
}
```

3.1.2 Procés de Backtracking

Aquest procés implica col·locar recursivament un número a la cel·la actual i verificar si és vàlid respecte a les restriccions de fila, columna i regió. Si col·locar el número compleix totes les restriccions, es procedeix a la cel·la següent. Si no es troba una configuració vàlida, es realitza un backtrack i es prova amb el següent número possible.

Listing 2: Procés de Backtracking del Validador

```
public boolean backtrackValidation(int n, int i, int j, int[][] rows, int[][] cols, int[]
    ↪ resDefReg, int[] resReg, Map<Cell<Integer, Integer>, InfoCoord<Integer, Integer, Integer
    ↪ >> regionMap, int[][] region, int[] actNReg) {
    // Algorisme de backtracking
}
```

3.1.3 Terminació

L'algoritme finalitza amb èxit quan es troba una solució vàlida, o falla després d'haver explorat totes les configuracions possibles sense èxit.

Listing 3: Terminació del Validador

```
if (i == n) {
    System.out.println("Kenken␣SOLUCIONADO:␣");
    Utils.printKenken(kenken, n);
    return true;
}
```

3.2 Conclusions del Validador

La implementació del validador de KenKen demostra com un enfocament sistemàtic de backtracking pot ser utilitzat per abordar problemes complexos basats en restriccions. Aquest algoritme és capaç de trobar solucions vàlides eficientment, sempre que es proporcionin les estructures de dades i les restriccions correctes. La seva modularitat permet adaptar-lo a diverses configuracions de KenKen, garantint la seva aplicabilitat en una àmplia gamma de situacions.

4 Classe Generator

La classe `Generator` és responsable de generar taulers de KenKen. Inclou mètodes per crear un tauler inicial i dividir-lo en regions, assignant operacions aritmètiques a cada regió segons la dificultat especificada.

Atributs Principals:

- `int size`: mida del tauler.
- `List<Operator> operators`: llista d'operadors aritmètics disponibles.

Mètodes Principals:

- `int[][] generate(int N)`: genera un tauler de mida N.
- `boolean validate(int[][] board)`: valida el tauler generat.
- `int[][] generateBoard(int N)`: crea un tauler inicial.
- `void generateRegions(...)`: divideix el tauler en regions i assigna operacions.
- `Integer calcOperation(...)`: calcula el resultat de l'operació aritmètica d'una regió.
- `String generateKenken(int N, int difficulty, boolean[] operands)`: genera el tauler complet de KenKen.

El mètode `generate` crea un tauler de mida $N \times N$. El mètode `validate` comprova si el tauler generat compleix amb les regles de KenKen. El mètode `generateBoard` inicialitza un tauler buit i l'omple amb números seguint les regles del KenKen. El mètode `generateRegions` divideix el tauler en regions i assigna operacions aritmètiques a cada regió segons la dificultat especificada. El mètode `calcOperation` calcula el resultat de l'operació aritmètica d'una regió. Finalment, el mètode `generateKenken` genera el tauler complet de KenKen segons la mida i la dificultat especificades, i retorna el tauler en format de text.

4.1 Funcionament General de l'Algoritme de Generació

4.1.1 Inicialització i Configuració Inicial

El procés de generació comença amb la creació d'un tauler buit i l'emplenament inicial de números, assegurant-se que compleixen les regles bàsiques de KenKen.

Listing 4: Inicialització del Generador

```
private void fillBoard(int[][] board) {
    int size = board.length;
    int[][] rows = new int[size][size];
    int[][] columns = new int[size][size];
    randomSizeStart(board, size, rows, columns);
    BooleanWrapper finished = new BooleanWrapper(false);
    completeBoard(board, size, 0, 0, rows, columns, finished);
}
```

4.1.2 Procés de Generació de Regions

Aquest pas implica la divisió del tauler en regions, assignant una operació aritmètica a cada regió segons els paràmetres de dificultat especificats. També es verifica que cada regió compleixi amb les restriccions aritmètiques.

Listing 5: Generació de Regions

```
public void generateRegions(int n, int r, List<Region> regions, boolean[] oper, int[][] board)
    ↪ {
    int[][] visited = new int[n][n];
    List<Integer> operDisp = new ArrayList<>();
    for (int i = 0; i < oper.length; ++i) {
        if (oper[i]) {
            operDisp.add(i+1);
        }
    }
    // Divisi  del tauler en regions i assignaci  d'operacions
}
```

4.1.3 Càlcul d'Operacions

El mètode `calcOperation` calcula el resultat de l'operació aritmètica assignada a una regió. Aquest mètode és crucial per assegurar-se que les regions compleixin amb les restriccions aritmètiques definides.

Listing 6: Càlcul d'Operacions

```
public Integer calcOperation(List<Cell<Integer, Integer>> cellList, int[][] board, int op) {
    int res = 0;
    // C lcul del resultat de l'operaci
    return res;
}
```

4.1.4 Generació del KenKen

Finalment, el mètode `generateKenken` genera el tauler de KenKen complet, incloent la divisió en regions i l'assignació d'operacions segons la dificultat especificada.

Listing 7: Generació del KenKen

```
public String generateKenken(int N, int difficulty, boolean[] operands) {
    int R = 1;
    if (difficulty == 1) {
        R = (int) (N*N)/2 + 2;
    } else if (difficulty == 2) {
        R = (int) (N*N)/2;
    } else if (difficulty == 3) {
        R = (int) (N*N)/2 - 4;
```

```
    } else {
        throw new IllegalArgumentException("Dificultad no v lida.");
    }
    int[][] board = generateBoard(N);
    System.out.println("TABLERO GENERADO DE FORMA ALEATORIA: ");
    Utils.printBoard(board);
    List<Region> regions = new ArrayList<>(R);
    generateRegions(N, R, regions, operands, board);
    System.out.println("REGIONES GENERADAS EN FORMA DE .TXT ESPECIFICADO: ");
    String newKenken = Utils.parseToString(N, R, regions);
    return newKenken;
}
```

4.2 Conclusions del Generador

El generador de KenKen és un component crucial per a la creació de puzles desafiants i entretinguts. La seva capacitat per generar taulers aleatoris i dividir-los en regions amb operacions aritmètiques específiques permet crear una àmplia varietat de puzles. L'algoritme de generació assegura que cada tauler compleixi les restriccions necessàries, proporcionant una experiència de joc coherent i satisfactòria.

5 Conclusió General

L'algorisme de backtracking per a KenKen és un exemple potent de com es poden aplicar tècniques de cerca exhaustiva i poda per resoldre puzles complexos basats en restriccions. L'optimització contínua i el refinament d'aquest algorisme poden portar a millores significatives en eficiència i temps d'execució, facilitant la resolució de puzles més grans o de major complexitat aritmètica.

Tant el validador com el generador de KenKen demostren la importància de les estructures de dades adequades i dels algorismes ben dissenyats en la resolució de problemes complexos. La seva implementació modular i adaptable assegura que es puguin aplicar a una àmplia gamma de configuracions de KenKen, proporcionant eines poderoses tant per a la validació com per a la generació de puzles.