Aufgabe 5: Rominos

Team-ID: 00619

Team-Name: ncrypted

Bearbeiter/-innen dieser Aufgabe:
Oliver Schirmer, Konrad Walter

22. November 2019

Inhaltsverzeichnis

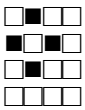
Lösungsidee

Der Ansatz basiert auf dynamischer Programmierung. Es wird der kleinste Fall (n = 2), der alle Vorgaben erfüllt, betrachtet: Um zu gewährleisten, dass sich mindestens zwei Quadrate diagonal berühren, müssen diese in einer solchen Diagonale angeordnet sein.

Die zweite Bedingung ist, dass es kein drittes Quadrat geben darf, welches eine gemeinsame Seite mit den beiden eben genannten Quadraten aufweist. Damit lassen sich in aufsteigender Reihenfolge für alle größeren n die Kombinationsmöglichkeiten der Rominos bilden. Es wird über alle bei n-1 gebildeten Rominos iteriert und bei dem Quadrat, welches bei der aktuellen Kombination als letztes hinzugefügt wurde, alle Möglichkeiten betrachtet, wie das neue Quadrat angelegt werden könnte. Dies sind einerseits die vier mit einer Seite anliegenden Quadrate und 4 diagonal mit einer Ecke anliegenden Quadrate. Dabei muss sichergestellt werden, dass das Feld nicht bereits belegt ist und die zweite Bedingung nicht verletzt wird, d.h. die Zerstörung der Diagonale, die durch die ersten beiden Quadrate erstellt wurde, muss verhindert werden. Dafür wird bei jeder neuen Kombinationsmöglichkeit überprüft, ob das neu anzulegende Quadrat auf einem eben solchen verbotenen Feld liegt und ggf. als Kombinationsmöglichkeit verworfen.

Umsetzung

Die Rominos werden als eine n*n Matrix gespeichert. Dabei werden jedoch nicht alle Felder gespeichert, sondern eine ArrayList angelegt (welche die Reihen darstellt), die n HashSets enthält. Diese wiederum enthalten mehrere Zahlen, die die Spaltennummern in der entsprechenden Reihe wiederspiegeln, welche ein Quadrat enthalten. Dadurch wird einerseits der Speicherbedarf minimiert und andererseits eine Dynamik hinsichtlich der Spaltenindexes gewährleistet, welche für den Algorithmus nötig ist. In folgendem Fall bspw. würde das Quadrat in der ersten Spalte einen negativen Index erhalten, womit eine Liste nicht umgehen könnte.



Aus diesem Grund müssen die Indexes vor der Ausgabe normalisiert werden, wobei der minimale Spaltenindex minColumn ermittelt wird und alle Spaltenindexes um minColumn nach links verschoben werden.

Zudem gibt es eine gleichartige Matrix, die die Felder speichert, für die weitere Quadrate verboten sind – Felder die bspw. die Diagonale des 2-Rominos zerstören würden – wodurch sichergestellt ist, dass die Anforderungen an ein Romino in jedem Falle erfüllt werden.

Es wird also von zwei bis einschließlich n iteriert, wobei beim ersten Fall n = 2 die Felder (0|0) und (1|1) standardmäßig festgelegt sind. Daraufhin wird über jede bereits erstellte Kombination iteriert und alle umliegenden Felder geprüft. Wurde eine gültige neue Kombination gefunden, wird diese in einer separaten Liste gespeichert. Diese Liste ersetzt am Ende die bisherige Liste, welche alle Kombinationen enthält. Dies ist zwar speichertechnisch, als auch performancetechnisch nicht sehr effektiv, vor allem im Vergleich zu In-Place Kombinationsbildung. Diese lässt sich aber hier nicht anwenden, da nicht nur ein Fall gefunden werden muss, sondern im Worst-Case sieben und die Ausgangskombination eventuell schon bei der zweiten gefundenen Kombination verändert wurde.

Beim Überprüfen von neuen Kombinationen muss zudem darauf geachtet werden, dass die Reihen einen gültigen Index haben, also größer gleich 0 und kleiner gleich n-1 sind. Ein weiterer sehr wichtiger Aspekt um sicherzustellen, dass keine Rominos entstehen, die durch Spiegelung an der Diagonalen von (0|0) bis (n-1|n-1) gleich sind, findet sich im Zuge der Überprüfung eines neuen Quadrates. Es dürfen keine neuen Quadrate, die oberhalb dieser Diagonalen liegen, also deren Spaltenindex größer ist als der Zeilenindex, hinzugefügt werden, außer es existiert bereits eins auf entsprechendem gespiegeltem Feld.

Beispiele

n	Kombinationen
2	1
3	3
4	14
5	52
6	159

n	Kombinationen
7	441
8	1220
9	3429
10	9845

4-Rominos:

1: □■□□ ■□■□ □■□□ □□□□	2: □■□□ □□■□ ■■□□ □□□□	3: □■□□ □□■□ □■□□ ■□□□	4: ■□□□ □■□□ ■□□□ ■□□□	5: ■□□□ □■□□ ■■□□ □□□□	6: ■□□□ □■□□ ■□□□ □■□□	7: ■□□□ □■□□ □■□□ ■□□□
8: ■□□□ □■□□ □■□□ □■□□	9: ■□□□ □■■□ □■□□ □□□□	10: ■□□□ □■□□ □■■□ □□□□	11: ■□□□ □■□□ □□■□ □□□□	12: ■□□□ □■□□ □□■□ □■□□	13: ■□□□ □■□□ □□■□ □□■□	14: ■□□□ □■□□ □□■□ □□□■

5-Rominos:

1: ■□■□□ □■□■□ □□■□□ □□□□□ □□□□□	2: □□■□□ ■■□■□ □□■□□ □□□□□ □□□□□	3: □□■□□ □■□■□ ■□■□□ □□□□□ □□□□□	4: ■■□□□ ■□■□□ □■□□□ □□□□□ □□□□□	5: □■□□□ ■□■□□ ■■□□□ □□□□□ □□□□□	6: □□■□□ ■□□■□ □■■□□ □□□□□ □□□□□	7: □□■□□ □□□■□ ■■■□□ □□□□□ □□□□□	8: □□■□□ □□□■□ □■■□□ ■□□□□ □□□□□

9: □■□□□ □□■□□ ■■□□□ ■□□□□ □□□□□	10: □■□□□ □□■□□ ■■□□□ □■□□□ □□□□□	11: □□■□□ □□□■□ ■□■□□ □■□□□ □□□□□	12: □□■□□ □□□■□ □□■□□ ■■□□□ □□□□□	13: □□■□□ □□□■□ □□■□□ □■□□□ ■□□□□	14: □■□□□ □□■□□ □■□□□ ■□□□□ ■□□□□	15: □■□□□ □□■□□ □■□□□ ■■□□□ □□□□□	16: □■□□□ □□■□□ □■□□□ ■□□□□ □■□□□
17: □■□□□ □□■□□ □■□□□ □■□□□ ■□□□□	18: ■□□□□ □■□□□ ■□□□□ ■□□□□ ■□□□□	19: ■□□□□ □■□□□ ■□□□□ ■■□□□ □□□□□	20: ■□□□□ □■□□□ ■□□□□ ■□□□□ □■□□□	21: ■□□□□ □■□□□ ■■□□□ □■□□□ □□□□□	22: ■□□□□ □■■□□ ■■□□□ □□□□□ □□□□□	23: ■□□□□ □■□□□ ■■□□□ □□■□□ □□□□□	24: ■□□□□ □■□□□ ■□□□□ □■□□□ ■□□□□
25: ■□□□□ □■□□□ ■□□□□ □■□□□ □■□□□	26: ■□□□□ □■□□□ ■□□□□ □■■□□ □□□□□	27: ■□□□□ □■□□□ ■□□□□ □■□□□ □□■□□	28: □■□□□ □□■□□ □□■□□ □■□□□ ■□□□□	29: ■□□□□ □■□□□ □■□□□ ■□□□□ ■□□□□	30: ■□□□□ □■□□□ □■□□□ ■□□□□ □■□□□	31: ■□□□□ □■□□□ □■□□□ □■□□□ ■□□□□	32: ■□□□□ □■□□□ □■□□□ □■□□□ □■□□□
33: ■□□□□ □■□□□ □■□□□ □■■□□ □□□□□	34: ■□□□□ □■□□□ □■□□□ □■□□□ □□■□□	35: ■□□□□ □■□□□ □■■□□ □□■□□ □□□□□	36: ■□□□□ □■□□□ □■■□□ □□□■□ □□□□□	37: ■□□□□ □■□□□ □■□□□ □□■□□ □■□□□	38: ■□□□□ □■□□□ □■□□□ □□■□□ □□■□□	39: ■□□□□ □■□□□ □■□■□ □□■□□ □□□□□	40: ■□□□□ □■□□□ □■□□□ □□■■□ □□□□□
41: ■□□□□ □■□□□ □■□□□ □□■□□ □□□■□	42: ■□□□□ □■□□□ □□■□□ □■□□□ ■□□□□	43: ■□□□□ □■□□□ □□■□□ □■□□□ □■□□□	44: ■□□□□ □■□□□ □□■□□ □■□□□ □□■□□	45: ■□□□□ □■□□□ □□■□□ □□■□□ □■□□□	46: ■□□□□ □■□□□ □□■□□ □□■□□ □□■□□	47: ■□□□□ □■□□□ □□■■□ □□■□□ □□□□□	48: ■□□□□ □■□□□ □□■□□ □□■■□ □□□□□
49: ■□□□□ □■□□□ □□■□□ □□■□□ □□□■□	50: ■□□□□ □■□□□ □□■□□ □□□■□ □□■□□	51: ■□□□□ □■□□□ □□■□□ □□□■□ □□□■□	52: ■□□□□ □■□□□ □□■□□ □□□■□ □□□□■				

Quellcode

```
package de.ncrypted.rominos;

public class Rominos {

    private static int n = -1;
    private static List<Romino> combos = new ArrayList<>();

    public static void main(String[] args) {
       // read n ...

        executeAlgorithm();

        System.out.println("---Ergebnis---");
        System.out.println("Kombinationen: " + combos.size());
        for (int i = 0; i < combos.size(); i++) {
            Romino combo = combos.get(i);
            combo.normalize();
```

```
            System.out.println((i + 1) + ":");
            System.out.println(combo);
        }
    }

    private static void executeAlgorithm() {
        List<Set<Integer>> blocked = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            blocked.add(new HashSet<>());
        }
        for (int i = 2; i <= n; i++) {
            // dynamical programming approach - start case n=2 ->
diagonal
            if (i == 2) {
                combos.add(new Romino().add(0, 0).add(1, 1));
                // block square that`d destroy the diagonal
                blocked.get(0).add(1);
                blocked.get(1).add(0);
                continue;
            }
            List<Romino> newCombos = new ArrayList<>();
            for (int j = 0; j < combos.size(); j++) {
                Romino romino = combos.get(j);
                for (int columnShift = -1; columnShift <= 1;
columnShift++) {
                    int column = romino.lastColumn + columnShift;
                    for (int rowShift = -1; rowShift <= 1;
rowShift++) {
                        int row = romino.lastRow + rowShift;
                        if(row < 0 || row > n-1
                                // prevent mirrored duplicate
rominos
                                || (column > row && !
romino.getMatrix().get(column).contains(row))
                                ||
blocked.get(row).contains(column)
                                ||
romino.getMatrix().get(row).contains(column)) {
                            continue;
                        }
                        newCombos.add(romino.copy().add(row,
column));
                    }
                }

blocked.get(romino.lastRow).add(romino.lastColumn);
            }
            combos = newCombos;
        }

    }

    private static class Romino {
        private List<Set<Integer>> pos;
        public int lastRow;
        public int lastColumn;

        public Romino() {
            ...
        }
```

```
            public Romino copy() {
                 ...
            }

            public Romino add(int row, int column) {
                ...
            }

            public List<Set<Integer>> getMatrix() {
                ...
            }

            public void normalize() {
                int min = 0;
                for (Set<Integer> row : pos) {
                    for (Integer column : row) {
                        min = Math.min(min, column);
                    }
                }
                for (int j = 0; j < pos.size(); j++) {
                    Set<Integer> row = pos.get(j);
                    Set<Integer> shifted = new HashSet<>(row.size());
                    for (Integer x : row) {
                        shifted.add(x - min);
                    }
                    pos.set(j, shifted);
                }
            }

            public String toString() {
                ...
            }
        }
    }
```