Aufgabe 5: Rominos

Team-ID: 00062

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker

24. November 2019

Inhalt

Lösungsidee

Ein n-Romino kann – wie auch ein n-Pentomino – immer aus einem (n-1)-Romino gebildet werden, dem ein weiteres Quadrat angefügt wurde (für n > 1). Das macht sich der Algorithmus zunutze, indem zuerst das einzige 2-Romino gebildet wird, daraus dann alle 3-Rominos und so weiter.

Effiziente Speicherung

Ein Romino wird als zweidimensionales Array dargestellt, dessen Elemente angeben, ob sich an dieser Stelle ein Quadrat befindet oder nicht. Um möglichst wenig Speicherplatz zu verbrauchen, wird dieses Array in einer Binärzahl gespeichert (im Folgenden „Binärzahl-Array“), bei der jede Ziffer ein Element darstellt. Zusätzlich wird in einem Tupel die Größe des Arrays gespeichert. Aus dem 3-Romino, das auf dem Aufgabenblatt ganz links abgebildet ist, wird das folgende Array:

5	4
3	2
1	0

`010110` ist die zugehörige Binärzahl mit der Größe `(3, 2)`. Dargestellt sind auch die Indizes, die bei 0 in der rechten unteren Ecke beginnen.
Damit Rominos, die durch Drehung oder Spiegelung zur Deckung gebracht werden können,[1] leichter erkannt werden, werden Rominos jedoch immer so gespeichert, dass das Array breiter als hoch oder quadratisch ist. Außerdem werden, nachdem ein neues Romino erstellt wurde, alle Möglichkeiten für die Drehung und Spiegelung ausprobiert, und das Romino wird als die größte Binärzahl, die entsteht, gespeichert. Im Beispiel müsste die Größe also `(2, 3)` sein und das Romino wird um 90° nach rechts gedreht, wodurch die größtmögliche Binärzahl, `110001`, entsteht. Das Array sieht dann folgendermaßen aus:

5	4	3
2	1	0

Randfelder berechnen

Alle leeren Felder in einem Array, die sich neben einem Quadrat befinden (auch diagonal), sind Randfelder. Um aus einem n-Romino neue (n+1)-Rominos zu bilden, wird nacheinander an jede Position eines Randfelds ein Quadrat gesetzt. Um alle Randfelder zu finden, werden alle Felder durchgegangen und wenn das Feld ein Quadrat enthält, werden alle leeren Felder aus den insgesamt acht umliegenden Feldern zu der Liste der Randfelder für dieses Romino hinzugefügt. Außerdem kann beim Durchgehen der umliegenden Felder geprüft werden, ob es eine nur-diagonale

[1] Da es im Array keine leeren Zeilen oder Spalten gibt, ist eine Verschiebung nicht notwendig.

Verbindung gibt und das Array damit wirklich ein Romino darstellt.[2] Diese gibt es genau dann, wenn es ein diagonales Feld mit Quadrat zwischen zwei leeren Feldern gibt.

Umsetzung

Das Programm wurde in Python 3.7 geschrieben und ist in der Datei aufgabe5.py zu finden.
Um Binärzahl-Arrays zu drehen und zu spiegeln, sind verschiedene Bit-Operationen nötig. Diese werden im Folgenden besprochen.

Bit-Operationen

Drehung um 90°

Eine Drehung um 90° ist dann nötig, wenn das Romino höher als breit ist. Um das Romino zu drehen, wird für jedes Feld seine neue Position berechnet. Die Position eines Feldes ist eine Zahl, und zwar die Stelle des Feldes in der Binärzahl, wobei links Position 0 ist. Wie im Folgenden veranschaulicht, berechnet sich die neue Position mit (alte Position · 2 + 1) mod (Länge der Binärzahl + 1). w und h sind die Breite und Höhe des Arrays.

5	4	3
2	1	0

5	4
3	2
1	0

alter Index	neuer Index $(i \cdot 2 + 1)\ mod\ (wh + 1)$
0	$(0 \cdot 2 + 1)\ mod\ 7 = 1$
1	$(1 \cdot 2 + 1)\ mod\ 7 = 3$
2	$(2 \cdot 2 + 1)\ mod\ 7 = 5$
3	$(3 \cdot 2 + 1)\ mod\ 7 = 0$
4	$(4 \cdot 2 + 1)\ mod\ 7 = 2$
5	$(5 \cdot 2 + 1)\ mod\ 7 = 4$

Das neue Binärzahl-Array ist zunächst 0. Für jedes Element im alten Array, das ein Quadrat enthält, wird die neue Position berechnet und das Quadrat in die neue Binärzahl per OR-Operation eingefügt. Die 1, die das Quadrat darstellt, wird dabei mit einer Bitverschiebung an die richtige Stelle verschoben: `1 << neuerIndex`.

Drehung um 180°

Obwohl die Drehung um 180° äquivalent zu einer horizontalen Spiegelung gefolgt von einer vertikalen Spiegelung ist, wird sie hier mit einer einfacheren Transformation implementiert. Der Index wird dabei sozusagen „umgedreht“, aus dem früher kleinsten Index wird der größte und aus dem größten der kleinste.

5	4	3
2	1	0

5	4	3
2	1	0

alter Index	neuer Index $wh - 1 - i$
0	$5 - 0 = 5$
1	$5 - 1 = 4$
2	$5 - 2 = 3$
3	$5 - 3 = 2$
4	$5 - 4 = 1$
5	$5 - 5 = 0$

Vertikale Spiegelung

[2] Durch Anfügen eines Quadrats an ein Romino können auch Nicht-Rominos entstehen.

Die vertikale Spiegelung berechnet den neuen Index mit (Breite - 1 - alter Index) mod (Länge der Binärzahl)

5	4	3
2	1	0

5	4	3
2	1	0

alter Index	neuer Index $(w-1-i)\ mod\ (hw)$
0	$(2-0)\ mod\ 6 = 2$
1	$(2-1)\ mod\ 6 = 1$
2	$(2-2)\ mod\ 6 = 0$
3	$(2-3)\ mod\ 6 = 5$
4	$(2-4)\ mod\ 6 = 4$
5	$(2-5)\ mod\ 6 = 3$

Horizontale Spiegelung

Die horizontale Spiegelung wird durchgeführt, indem bei jeder Reihe der alten Binärzahl die Höhe per Bit-Operationen verändert wird. Aus der x. Reihe wird die $(h-1-x)$. Reihe.

0	0	0
1	1	1

0	0	0
1	1	1

Anfügen von Reihen/Spalten

Wenn ein Quadrat außerhalb der Binärzahl angefügt wird, muss die Größe verändert werden. Wird eine Reihe oben angefügt, ist keine Bit-Operation nötig, weil die Reihe von – zunächst Nullen – sowieso vor der Binärzahl angefügt werden müsste. Um eine Reihe unten anzufügen, wird die Binärzahl einfach so viele Stellen nach links verschoben, wie das Feld breit ist. Für das Anfügen links oder rechts ist es nötig, eine neue Binärzahl zu erstellen, bei der per OR-Operation die einzelnen Reihen an der richtigen Stelle eingefügt werden.

Die Romino-Klasse

Die Klasse `Romino` speichert ein Romino, also das Array als Binärzahl (`Romino.array`) sowie die Größe als ein zwei Elemente enthaltendes Tupel (`Romino.shape`) und die Randfelder (`Romino.edges`). Die statische Methode `new` erstellt ein neues Romino aus einer Größe und dem zugehörigen Binärzahl-Array und gibt dieses zurück. Indem alle Möglichkeiten mit Drehung und Spiegelung ausprobiert werden,[3] erhält man die größtmögliche Binärzahl. Außerdem werden alle Randfelder gesucht und im Attribut und geprüft, ob es sich um ein valides Romino handelt, wie in der Lösungsidee beschrieben. Sollte es sich nicht um ein Romino handeln, wird `None` zurückgegeben.

Ein `Romino` ist hashable, damit es in einer Menge (`set`) gespeichert werden kann. Der Hashwert wird aus dem Array und der Form wie folgt gebildet: `(array << 8) + (shape[0] << 4) + shape[1]` (`<<` ist eine Bitverschiebung nach links). Ein Array kann maximal die Seitenlänge 10 haben, was maximal 4 Bit sind. Deshalb befindet sich die Größe in den letzten 8 Bit des Hashwerts, davor steht das Array mit variabler Länge. Auf diese Weise gibt es für jeden Hashwert auch wirklich nur ein Romino. Es ist hier keine gute Idee, die `hash()`-Built-In-Funktion von Python zu verwenden, beispielsweise indem der Hashwert so berechnet wird: `hash(array) + hash(shape)`. Damit ist nämlich nicht

[3] Bei Arrays mit unterschiedlicher Höhe und Breite gibt es zwei Möglichkeiten zum Drehen (0° und 180°); bei einem quadratischen Array gibt es vier Möglichkeiten. Da es pro Drehung drei mögliche Spiegelungen gibt (nicht gespiegelt, horizontal gespiegelt, vertikal gespiegelt; sowohl vertikal als auch horizontal ergeben 180°), gibt es für quadratische Arrays zwölf Möglichkeiten, für andere nur halb so viele.

auszuschließen, dass zwei eigentlich unterschiedliche Rominos denselben Hashwert haben. Die Implementierung, die die `hash()`-Funktion benutzt, liefert abhängig vom Betriebssystem und abhängig davon, ob die 64- oder 32-Bit-Version von Python benutzt wird, unterschiedliche Ergebnisse. Das liegt einerseits daran, dass `hash()` natürlich gleiche Werte für unterschiedliche Arrays/Größen zurückgeben kann und andererseits daran, dass Python bei einem Aufruf von `hash()` bei zu großer Länge Ziffern abschneidet.[4] Die `hash()`-Funktion ruft die `__hash__`-Methode (die Methode in Python, die einen Hashwert zurückgeben sollte) des übergebenen Objekts (hier also ein Objekt der Klasse `Romino`) auf, um den Hashwert zu erhalten, und dieser wird gegebenenfalls gekürzt. Da mit `set`s gearbeitet wird, ist das nicht weiter wichtig (`set` nutzt die `__hash__`-Methode), aber falls zwei Rominos verglichen werden sollen, sollten dabei nicht die Rückgabewerte der `hash()`-Funktion verglichen werden, sondern die der `__hash__`-Methode.
Der berechnete Hashwert wird im Attribut `Romino._hash` gespeichert und von der Methode `__hash__` zurückgegeben. Sobald ein Romino erstellt wurde, ändert sich der Hashwert nicht, weil alle Attribute nicht verändert werden (`Romino`s sind immutable).

Die Methode grow

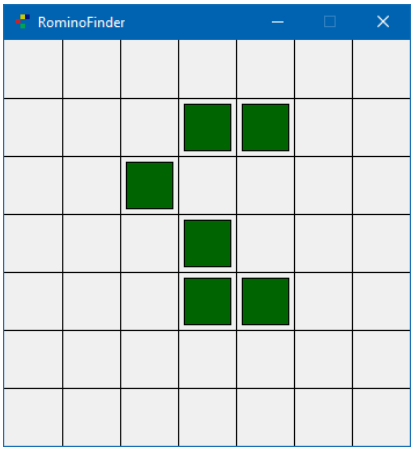
Ein `Romino`-Objekt hat die Methode `grow`. Diese Methode gibt eine Menge (`set`) aller Rominos zurück, die von diesem Romino durch Anfügen eines Quadrats gebildet werden können. Dafür wird für jedes Randfeld ein neues Array mit diesem Randfeld erstellt. Sollte für das Anfügen des Quadrats keine Vergrößerung des Arrays nötig sein, wird der entsprechende Bit einfach per OR-Operation auf 1 gesetzt. Sollte stattdessen eine Vergrößerung nötig sein, wird diese vorher mit der unter „Bit-Operationen“ beschriebenen Methode vorgenommen.

Das Hauptprogramm

Ausgehend von einem 1-Romino (welches eigentlich gar kein Romino ist, da diagonal verbundene Quadrate fehlen) wird die zweite Generation, bestehend aus dem einzigen 2-Romino, berechnet und dann alle folgenden. Für n-Rominos, bei denen $n < 6$ gilt, werden außerdem alle Rominos grafisch ausgegeben, für größere n nur die Anzahl. Um die noch benötigte Zeit anzuzeigen, wird – sofern es installiert ist – das tqdm-Modul[5] benutzt.

Der Romino-Finder und Überprüfung des Programms

Um das Programm möglichst einfach auf Fehler zu überprüfen, wurde ein Programm namens „RominoFinder“ entwickelt. Der*die Benutzer*in gibt zunächst ein, bis zu welcher Anzahl der Quadrate Rominos berücksichtigt werden sollen. Nach dem Suchen aller Rominos bis zu dieser Größe wird ein Feld präsentiert, dessen Zellen er*sie per Klick mit einem Quadrat versehen kann bzw. dieses wieder löschen kann. Sollte nach



[4] siehe https://docs.python.org/3/reference/datamodel.html#object.__hash__ . Der Rückgabewert von `hash()` ist mit 32-Bit üblicherweise 4 Byte lang, mit 64-Bit 8 Byte.
[5] github.com/tqdm/tqdm

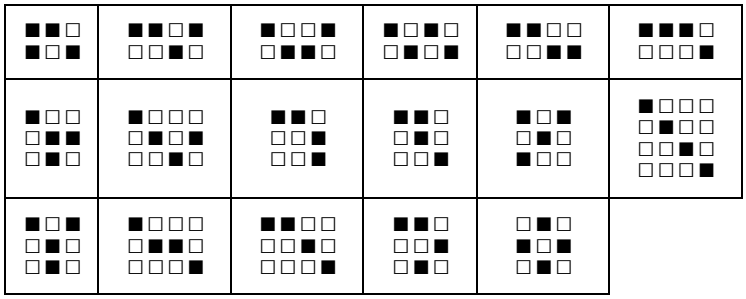
einem Klick ein valides Romino entstanden sein,[6] wird geprüft, ob dieses bereits gefunden wurde. Ist das nicht der Fall, erscheint eine entsprechende Meldung. Das Python-Skript dazu befindet sich in der Datei `romino_finder.py`. Es werden die Module numpy und PySide2 benötigt. Wenn das tqdm-Modul installiert ist, wird außerdem der Fortschritt der Berechnungen ausgegeben.

Das Herumprobieren mit dem Romino-Finder stellt allerdings noch nicht sicher, dass nicht zu viele Rominos gefunden wurden. Unter der Annahme, dass gleiche Rominos nicht doppelt vorkommen, kann es nur noch passieren, dass sich ein Romino unter n-Rominos findet, das aber nicht aus n Quadraten besteht. Um dies auszuschließen, wurden für 6-, 7- und 8-Rominos die Quadrate gezählt und überprüft, ob es sechs, sieben bzw. acht sind. Der Programmteil dazu befindet sich im aufgabe5.py-Skript ganz unten und ist auskommentiert.

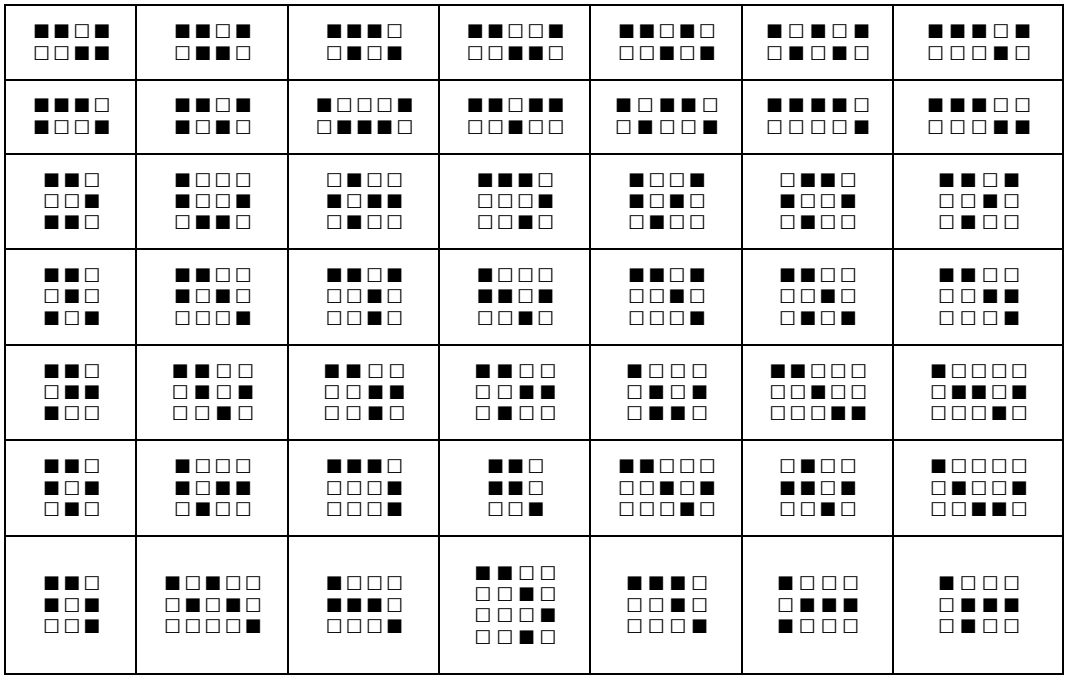
Beispiele

Zunächst die grafische Ausgabe für die 17 4-Rominos und 82 5-Rominos. Die Ausgabe ist auch in der Datei output.txt zu finden.

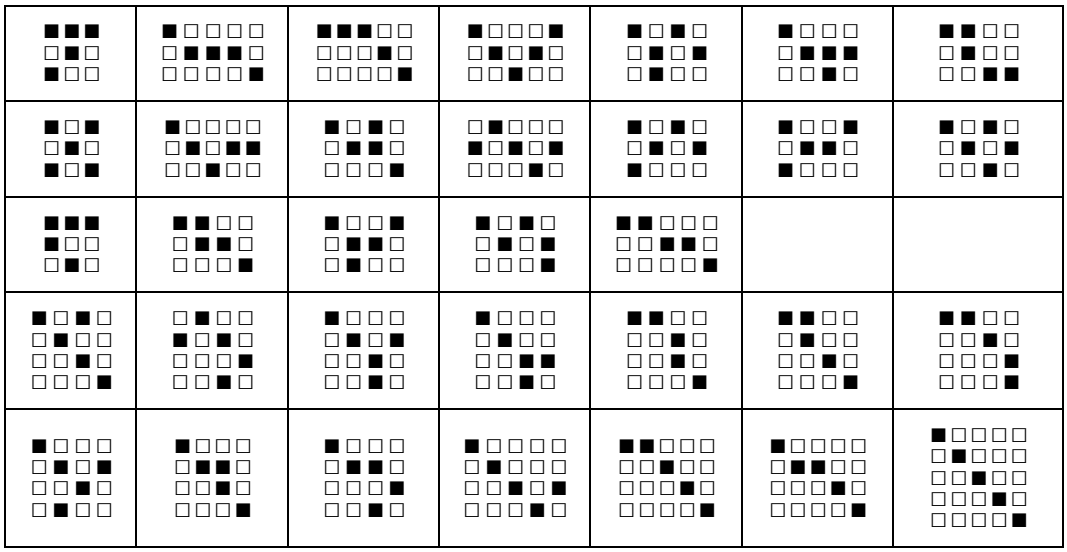
4-Rominos



5-Rominos



[6] Ist ein valides Romino entstanden, färben sich die Quadrate grün, andernfalls rot, wenn es Lücken zwischen den Quadraten gibt, und schwarz, wenn die Quadrate zwar zusammenhängend sind, aber nicht die Bedingung für ein Romino erfüllen.



Anzahlen der Rominos

n	Anzahl
3	3
4	17
5	82
6	489
7	2924
8	18406
9	116883
10	753905

Die Anzahl der Rominos wächst ungefähr exponentiell. Eine Regression ergab folgende Formel für die ungefähre Anzahl von n-Rominos: $e^{-4{,}3573+1{,}7768n}$.

Quellcode

Bei Quellcodeauszügen ist zu beachten, dass Typangeben für Parameter in Python vom Interpreter ignoriert werden. Sie dienen nur der Dokumentation und der besseren Erkennung durch IDEs.

Bit-Operationen

Die oben beschriebenen Bit-Operationen wurden folgendermaßen implementiert; dies sollte selbsterklärend sein.

```
def rotate90(shape: Tuple[int, int], array: int):
    """Drehung 90°"""
    array_size = shape[0]*shape[1]
    array_size_plus_1 = array_size + 1
    height_minus_1 = shape[0] - 1
    x = 0
    for n in range(array_size):
        new_n = (n + (height_minus_1*(n+1))) % array_size_plus_1
        if (array >> n) & 1:
            x |= 1 << new_n
    return x

def rotate180(shape: Tuple[int, int], array: int):
    """Drehung 180°"""
```

```
    array_size = shape[0] * shape[1]
    array_size_minus_1 = array_size - 1
    x = 0
    for n in range(array_size):
        new_n = array_size_minus_1 - n
        if (array >> n) & 1:
            x |= 1 << new_n
    return x

def flip_lr(shape: Tuple[int, int], array: int):
    """Vertikale Spiegelung (links-rechts vertauschen)"""
    width = shape[1]
    new_array = 0
    for y in range(shape[0]):
        for x in range(width):
            y_pos = y*width
            n = y_pos + x
            new_n = y_pos + (width - 1 - x)
            if (array >> n) & 1:
                new_array |= 1 << new_n
    return new_array

def flip_ud(shape: Tuple[int, int], array: int):
    """Horizontale Spiegelung (unten-oben vertauschen)"""
    height = shape[0]
    height_minus_1 = height - 1
    width = shape[1]
    x = 0
    row_with_ones = (2**width - 1)
    for y in range(height):
        row = (array >> (y*width)) & row_with_ones
        new_y = height_minus_1 - y
        x |= row << new_y*width
    return x

def add_right(shape: Tuple[int, int], array: int):
    """Reihe rechts anfügen"""
    x = 0
    height = shape[0]
    width = shape[1]
    row_with_ones = (2 ** width - 1)
    for y in range(height):
        row = (array >> (y * width)) & row_with_ones
        x |= row << (y * width + y + 1)
    return x

def add_left(shape: Tuple[int, int], array: int):
    """Reihe links anfügen"""
    x = 0
    height = shape[0]
    width = shape[1]
    row_with_ones = (2 ** width - 1)
    for y in range(height):
        row = (array >> (y * width)) & row_with_ones
        x |= row << (y * (width + 1))
    return x
```

```
def add_bottom(shape: Tuple[int, int], array: int):
    """Reihe unten anfügen"""
    return array << shape[1]
```

Die Romino-Klasse

`__init__(shape: Tuple[int, int], array: int, edges: Iterable[Tuple[bool, Tuple[int, int]])`

Der Konstruktor bekommt die Größe, das Binärzahl-Array und die Liste mit Randfeldern übergeben, die in Attributen gespeichert werden. Für jedes Randfeld ist außerdem im ersten Element des Tupels gespeichert worden, ob es außerhalb des Arrays liegt, also eine Reihe/Spalte angefügt werden muss (siehe new-Methode).

`@staticmethod`
`new(shape: Tuple[int, int], array: int)`

Diese statische Methode erstellt ein neues Romino-Objekt und sollte statt des Konstruktors benutzt werden, da diese Methode auch die Randfelder berechnet. Zunächst wird das Binärzahl-Array aber richtig gedreht und gespiegelt:

```
if shape[0] == shape[1]:
    # Das Array ist quadratisch
    array_rot180 = rotate180(shape, array)
    array_rot90 = rotate90(shape, array)
    array_rot90_rot180 = rotate180(shape, array_rot90)
    array = max(
        array,
        array_rot180,
        flip_lr(shape, array),
        flip_lr(shape, array_rot180),
        flip_ud(shape, array),
        flip_ud(shape, array_rot180),
        array_rot90,
        array_rot90_rot180,
        flip_lr(shape, array_rot90),
        flip_lr(shape, array_rot90_rot180),
        flip_ud(shape, array_rot90),
        flip_ud(shape, array_rot90_rot180)
    )
else:
    if shape[0] > shape[1]:
        # Das Array ist höher als breit -> 90°-Drehung
        array = rotate90(shape, array)
        shape = (shape[1], shape[0])
    array_rot180 = rotate180(shape, array)
    array = max(
        array,
        array_rot180,
        flip_lr(shape, array),
        flip_lr(shape, array_rot180),
        flip_ud(shape, array),
        flip_ud(shape, array_rot180)
    )
```

Danach werden wie beschrieben die Randfelder berechnet und gleichzeitig überprüft, ob es Quadrate gibt, die nur diagonal verbunden sind (also das „Muster“ kein Quadrat – diagonales Quadrat – kein Quadrat). Wenn Koordinaten statt Indizes verwendet werden, entspricht (0, 0) dem Index 0 und (0, 1) dem Index 1.

```
edges = set()  # die Randfelder
is_valid = False  # wird auf True gesetzt, sobald eine nur-diagonale Verbindung gefunden wurde
# Alle Felder durchgehen
for y in range(shape[0]):
    for x in range(shape[1]):
        if (array >> (y * shape[1] + x)) & 1:
            # Das Feld enthält ein Quadrat
            y_down = y - 1
            y_up = y + 1
            x_down = x - 1
            x_up = x + 1
            if not is_valid:
                # Es muss noch überprüft werden, ob es ein valides Romino ist
                # in surrounding_squares werden Informationen über die umgebenden acht Quadrate
                # im Uhrzeigersinn gespeichert:
                # v: Feld enthält ein Quadrat und ist diagonal
                # e: Feld enthält ein Quadrat und liegt mit einer Seite am Feld (x, y) an
                # b: Feld enthält diagonal kein Quadrat
                # r: Feld liegt mit einer Seite am Feld (x, y) an und enthält kein Quadrat
                # für ein valides Romino muss also die Folge "rvr" in surrounding_squares sein
                surrounding_squares = ""
                # Alle umliegenden Felder im Uhrzeigersinn durchgehen (pos).
                # i ist 0 oder 1 und gibt an, ob es ein diagonales Feld ist oder nicht
                for pos, i in zip(
                        ((y_up, x_down), (y_up, x), (y_up, x_up),
                         (y, x_up),
                         (y_down, x_up), (y_down, x), (y_down, x_down),
                         (y, x_down)),
                        itertools.cycle([0, 1])):
                    if pos[0] == -1 or pos[1] == -1 or pos[0] >= shape[0] or
                                                                    pos[1] >= shape[1]:
                        # pos befindet sich außerhalb des Arrays, ist also auf jeden Fall ein
                        # Randfeld
                        # Die Randfelder werden mit der Information, ob es innerhalb des Arrays
                        # liegt, gespeichert, hier also False.
                        edges.add((False, pos))
                        surrounding_squares += ["b", "r"][i]
                    else:
                        if not (array >> (pos[0] * shape[1] + pos[1])) & 1:
                            # pos ist ein Feld ohne Quadrat, also ein Randfeld
                            edges.add((True, pos))
                            surrounding_squares += ["b", "r"][i]
                        else:
                            surrounding_squares += ["v", "e"][i]
                surrounding_squares += surrounding_squares[:3]  # "rvr" könnte auch von
                                        # (y, x_down) zu (y_up, x_down) (Start und Ende) vorkommen
                if "rvr" in surrounding_squares:
                    # "rvr" ist in surrounding_squares, es ist also ein Romino
                    is_valid = True
            else:
                # derselbe Code wie oben, nur ohne Überprüfung, weil bereits festgestellt wurde,
                # dass es ein valides Romino ist
```

```
                for pos, i in zip(
                        ((y_up, x_down), (y_up, x), (y_up, x_up),
                         (y, x_up),
                         (y_down, x_up), (y_down, x), (y_down, x_down),
                         (y, x_down)),
                        itertools.cycle([0, 1])):
                    try:
                        if pos[0] == -1 or pos[1] == -1 or pos[0] >= shape[0] or
                                                                    pos[1] >= shape[1]:
                            raise IndexError
                        if not (array >> (pos[0] * shape[1] + pos[1])) & 1:
                            edges.add((True, pos))
                    except IndexError:
                        # pos is outside _array
                        edges.add((False, pos))
if is_valid:
    return Romino(shape, array, edges)
return None
```

grow()

Diese Methode erstellt neue Rominos und gibt sie zurück, indem für die Randfelder Quadrate eingesetzt werden. Das sieht folgendermaßen aus:

```
new_rominos = set()  # alle neuen Rominos
for in_array, pos in self.edges:
    if in_array:
        # pos liegt im Array
        new_array = self.array
        new_shape = self.shape
        n = (pos[0] * new_shape[1]) + pos[1]  # die Position des neuen Quadrats
    else:
        # pos ist außerhalb des Arrays
        if pos[0] == -1:
            # y-Position ist zu klein (Position ist unter dem Array)
            if pos[1] == -1:
                # x-Position ist zu klein, pos ist an der unteren rechten Ecke des Arrays
                new_array = add_right((self.shape[0] + 1, self.shape[1]),
                                      add_bottom(self.shape, self.array))
                new_shape = (self.shape[0] + 1, self.shape[1] + 1)
                n = 0
            elif pos[1] >= self.shape[1]:
                # x-Position ist zu groß, pos ist an der unteren linken Ecke
                new_array = add_left((self.shape[0] + 1, self.shape[1]),
                                     add_bottom(self.shape, self.array))
                new_shape = (self.shape[0] + 1, self.shape[1] + 1)
                n = new_shape[1] - 1
            else:
                # x-Position ist im Array, pos ist unter dem Array
                new_array = add_bottom(self.shape, self.array)
                new_shape = (self.shape[0] + 1, self.shape[1])
                n = pos[1]
        elif pos[0] >= self.shape[0]:
            # y-Position ist zu groß (Position ist über dem Array)
            if pos[1] == -1:
                # x-Position ist zu klein, pos ist an der oberen rechten Ecke
                new_array = add_right(self.shape, self.array)
                new_shape = (self.shape[0] + 1, self.shape[1] + 1)
```

```
                n = (new_shape[0] - 1) * (new_shape[1])
            elif pos[1] >= self.shape[1]:
                # x-Position ist zu groß, pos ist an der oberen linken Ecke
                new_array = add_left(self.shape, self.array)
                new_shape = (self.shape[0] + 1, self.shape[1] + 1)
                n = new_shape[0] * new_shape[1] - 1
            else:
                # x-Position ist im Array, pos ist über dem Array
                new_array = self.array
                new_shape = (self.shape[0] + 1, self.shape[1])
                n = (new_shape[0] - 1) * new_shape[1] + pos[1]
        else:
            # y-Position ist im Array
            if pos[1] == -1:
                # x-Position ist zu klein, pos ist an der rechten Seite
                new_array = add_right(self.shape, self.array)
                new_shape = (self.shape[0], self.shape[1] + 1)
                n = pos[0]*new_shape[1]
            elif pos[1] >= self.shape[1]:
                # x-Position ist zu groß, pos ist an der linken Seite
                new_array = add_left(self.shape, self.array)
                new_shape = (self.shape[0], self.shape[1] + 1)
                n = pos[0]*new_shape[1] + pos[1]
            else:
                # die Position ist im Array, aber dann hätte in_array True sein müssen
                raise ValueError
    new_romino_array = new_array | 1 << n  # Quadrat einfügen
    romino = Romino.new(new_shape, new_romino_array)
    if romino is not None:
        new_rominos.add(romino)
return new_rominos
```

`__hash__()`

Damit Rominos in einer Menge (`set`) gespeichert werden können, müssen sie hashable sein. Hier wird der Hashwert zurückgegeben, der sich aus Größe und Array zusammensetzt und im Konstruktor einmalig berechnet wurde. Zur Berechnung und zu wichtigen Hinweisen zur Berechnung des Hashwerts siehe die Umsetzung.

`__equal__(other)`

Gibt zurück, ob die Hashwerte der beiden Rominos identisch sind.

Das Hauptprogramm

Wie beschrieben werden alle Rominos beginnend beim 1-Romino (das eigentlich gar kein Romino ist) berechnet:

```
romino1 = Romino.new((1, 1), 1)
generation = {romino1}
for i in range(2, 11):
    new_generation = set()
    for romino in tqdm(generation):
        new_generation.update(romino.grow())
    print("\n#################")
    print(f"# {i}-Rominos")
```

```
    print("Anzahl:", len(new_generation))
    if i < 6:
        for romino in new_generation:
            print(romino, end="\n\n")
    generation = new_generation
```

Alte Generationen, die für die Berechnung neuer Rominos nicht mehr benötigt werden, werden auch nicht gespeichert. `generation` ist ein `set`, das alle Rominos enthält, aus denen neue Rominos (`new_generation`) berechnet werden.