Junioraufgabe 2: Kacheln

Team-ID: 00062

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker

21. Oktober 2019

Inhalt

Lösungsidee

4 Bit können genau 16 Zahlen darstellen ($16=2^4$). Aus diesem Grund ist mit den 16 vorgegebenen Kacheln jede Möglichkeit für die 4 Bits abgedeckt. Somit ist es also nicht nötig, für eine freie Kachel alle Kacheln durchzuprobieren, sondern stattdessen wird für jedes leere Quadrat einzeln ein passendes Bit bestimmt.

Es gibt drei Fälle, in denen sich ein leeres Quadrat befinden kann, die durch das rechts abgebildete Beispiel verdeutlicht werden:

```
* *   * *
* *   * *
```

```
* *   0 0
* *   1 0
```

1. Im Beispiel grün: Ein Quadrat wird durch ein Quadrat einer anderen Kachel, bei dem bereits ein Bit gesetzt ist, berührt. In diesem Fall kann das Bit auf den gleichen Wert wie der des berührenden Quadrats gesetzt werden. Haben die zwei berührenden Quadrate aber unterschiedliche Bits, ist eine Ergänzung ohne Regelverletzung nicht möglich.
2. Im Beispiel rot: Ein berührendes Quadrat einer anderen Kachel wird erst später auf einen Wert gesetzt, sodass erst später ein Wert nach dem gleichen Schema wie beim 1. Fall ermittelt werden kann.
3. Im Beispiel blau: Das Bit eines Quadrats kann beliebig gewählt werden. Es ist jedoch sinnvoll, für diesen beliebigen Wert immer denselben Wert zu wählen, damit bei nebeneinanderliegenden Quadraten dieses dritten Typs keine Regelverletzung entsteht, wie es bei zufälliger Auswahl passieren würde.[1]

Der Algorithmus iteriert durch alle leeren Quadrate. Um die leeren Quadrate besonders effizient zu ermitteln und nicht alle Quadrate durchzugehen, um zu überprüfen, ob diese leer sind, werden die leeren Quadrate bereits beim Einlesen gespeichert. Für ein anfangs leeres Quadrat wird zunächst geprüft, ob es immer noch leer ist. Ist das der Fall, wird es besucht. Ein Quadrat kann, bevor es von dieser Hauptschleife besucht wurde, nicht mehr leer sein, weil beim Besuchen eines Quadrats umliegende Quadrate eventuell auch besucht werden.

Das Besuchen eines leeren Quadrats ähnelt dem Floodfill-Algorithmus. Um zu überprüfen, ob der Bit für diese Quadrat gesetzt werden kann und wenn ja auf welchen Wert, sind zwei der insgesamt vier benachbarten Quadrate relevant. Diese zwei benachbarten Quadrate sind die, die nicht zur selben Kachel gehören. Ist mindestens ein Bit der zwei Quadrate gesetzt, wird auch das Bit des gerade besuchten Quadrats auf diesen Wert gesetzt. Sind die zwei Bits der Quadrate unterschiedlich, existiert keine Lösung ohne Regelverletzung und wenn die zwei Quadrate leer sind, kann auch der Bit des besuchten Quadrats noch nicht gesetzt werden.[2] Wurde der Bit des Quadrats aber bereits gesetzt und ein Bit der benachbarten Quadrate war nicht gesetzt, wird dieses leere Quadrat nun besucht, weil das Bit für das leere Quadrat nun auch den Wert annehmen muss, auf den der Bit des gerade besuchten Quadrats gesetzt wurde. Auf diese Weise werden auch Bits von Quadraten des zweiten Falles gesetzt.

Zuletzt wird für alle Quadrate, die immer noch leer sind, ein vorher definierter Wert (0, 1 oder *) gesetzt.

[1] Es ist nicht notwendig, immer denselben Wert zu wählen. Wenn mehrere Quadrate dieses dritten Falles „zusammenhängen“, sollten die Bits aller Quadrate dieser Gruppe auf denselben Wert gesetzt werden. Der Wert kann aber von Gruppe zu Gruppe variieren.

[2] Das Quadrat gehört also entweder zum zweiten oder zum dritten Fall. Gehört es zum zweiten, wird das Bit später gesetzt, wenn ein leeres benachbartes Quadrat (siehe Satz nach dieser Fußnote) besucht wurde.

Umsetzung

Das Programm wurde mit Python 3 geschrieben. Getestet wurde es mit Python 3.7. Aufgrund der Verwendung von f-Strings ist das Programm nicht mit Python älter als Version 3.6 kompatibel.

Die Klasse `MapGenerator` liest eine Datei ein und füllt mit der Methode `fill_spaces` die leeren Quadrate. Die Landschaft wird in einem `numpy`-Array, dem Attribut `parts`, gespeichert. Die Elemente des Arrays sind die einzelnen Quadrate (nicht die Kacheln) und das Quadrat oben links hat den Index (0, 0). Eine 0 steht für Wasser, eine 1 für Land und eine 2 für einen Asterisk. Alle beim Einlesen ermittelten, leeren Quadrate werden im Attribut `original_empty_parts` gespeichert. Diese Liste wird nicht mehr verändert, während aus einer Kopie dieser Liste – `empty_parts` – Indizes für Quadrate, bei denen gerade ein Bit gesetzt wurde, entfernt werden. `empty_parts` wird später benötigt, um die Bits aller leer gebliebenen Quadrate zu setzen.
Die bereits genannte Methode `fill_spaces` ruft für alle leeren Quadrate die Methode `_set_bit` auf, die die Implementierung des an Floodfill orientierten Algorithmus aus der Lösungsidee darstellt. Die benachbarten Quadrate, die nicht zur selben Kachel gehören, werden mit dem Index des gerade besuchten Quadrats bestimmt: Wenn die y-Position gerade ist, befindet sich ein benachbartes, relevantes Quadrat über dem Quadrat, wenn die x-Position gerade ist, befindet sich ein benachbartes, relevantes Quadrat auf der linken Seite. Die Bits der zwei benachbarten, relevanten Quadrate werden überprüft und wenn der Bit für das Quadrat bestimmt werden konnte und eines der benachbarten Quadrate leer ist, wird die `_set_bit`-Methode für dieses leere Quadrat aufgerufen, um auch Bits von Quadraten des zweiten Falles zu setzen.

Beispiele

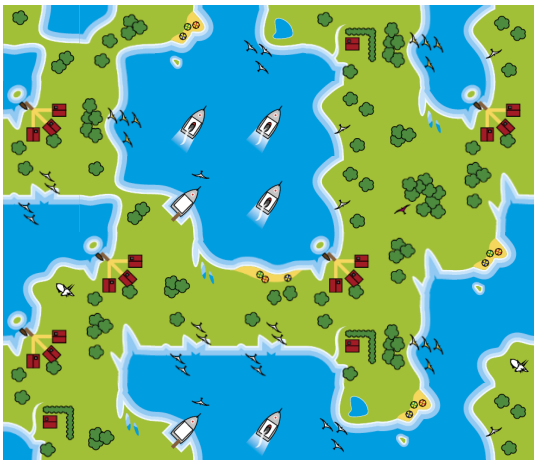
Die Lösungen der Beispiele sind im „solutions“-Ordner zu finden. Es gibt für jede Landschaft zwei Varianten: Bei der einen wurden Quadrate mit beliebig wählbarem Bit auf „Wasser“, bei der anderen auf „Land“ gesetzt. Die Folgenden Bilder sind jeweils die Variante mit „Wasser“. Bei der Ausgabe wurden beliebig wählbare Bits mit `*` kenntlich gemacht.
Die Farben in der Ausgabe werden vom Programm erzeugt. Grün steht für anfänglich leere Felder, ein rot eingefärbtes Bit bedeutet, dass dieses Bit nicht ohne Regelverletzung gesetzt werden kann.

`map`

```
1 0  0 *
0 1  1 0

0 1  1 0
* 1  1 1
```

map1



```
1 0   0 1   1 *   * 1   1 1   1 0   0 1
0 1   1 0   0 *   * 0   0 1   1 0   0 1

0 1   1 0   0 *   * 0   0 1   1 0   0 1
1 1   1 0   0 0   0 0   0 1   1 1   1 1

1 1   1 0   0 0   0 0   0 1   1 1   1 1
0 0   0 1   1 0   0 0   0 1   1 1   1 *

0 0   0 1   1 0   0 0   0 1   1 1   1 *
0 1   1 1   1 1   1 1   1 1   1 0   0 0

0 1   1 1   1 1   1 1   1 1   1 0   0 0
1 1   1 0   0 0   0 0   0 1   1 0   0 1

1 1   1 0   0 0   0 0   0 1   1 0   0 1
0 1   1 1   1 *   * *   * 0   0 0   0 1
```

map2

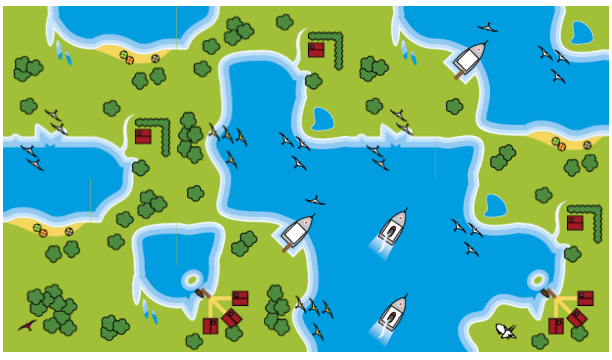
```
* *   * *   * *
* 0   0 0   0 *

* 0   0 0   0 *
* 1   1 0   0 *

* 1   1 0   0 *
* 0   0 0   0 *

* 0   0 0   0 *
* *   * *   * *
```

map3



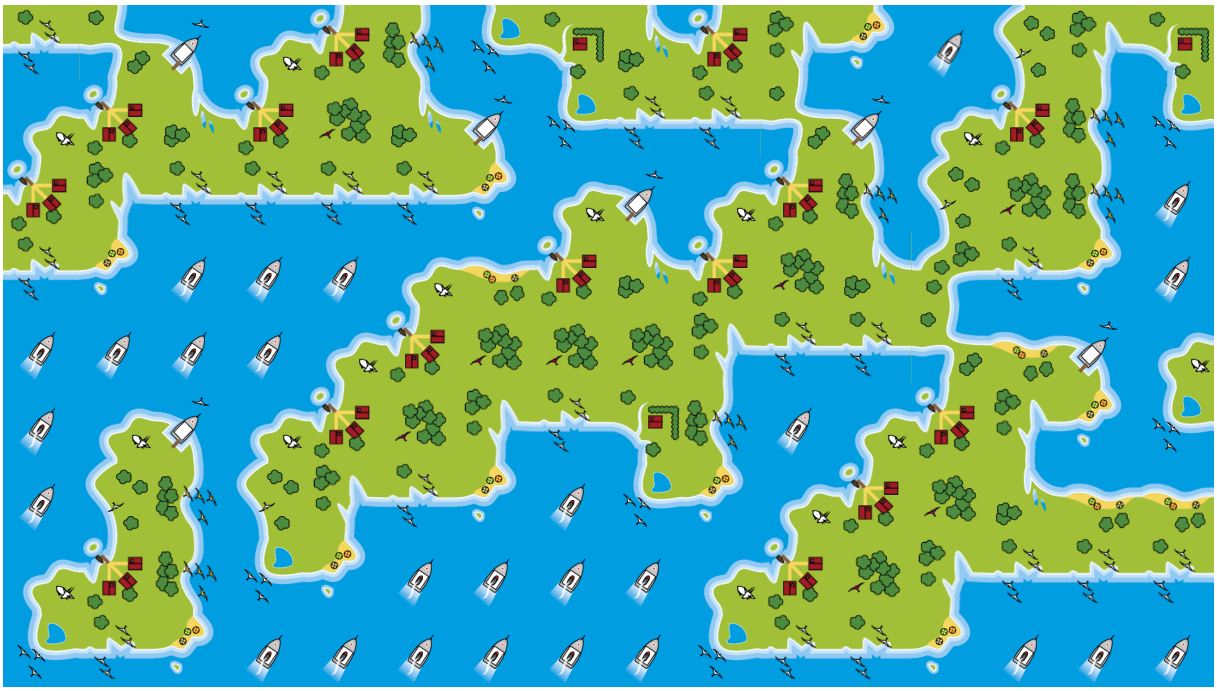
```
1 0   0 0   0 1   1 1   1 0   0 0   0 1
1 1   1 1   1 0   0 1   1 1   1 0   0 0

1 1   1 1   1 0   0 1   1 1   1 0   0 0
0 0   0 1   1 *   * *   * *   * 1   1 1

0 0   0 1   1 *   * *   * *   * 1   1 1
1 1   1 0   0 1   1 0   0 0   0 0   0 1

1 1   1 0   0 1   1 0   0 0   0 0   0 1
1 1   1 1   1 1   1 0   0 0   0 1   1 1
```

map4



```
1 1   1 0   0 0   0 0   0 1   1 0   0 1   1 1   1 0   0 1   1 1   1 0   0 0   0 1   1 1   1 1
0 0   0 1   1 0   0 1   1 1   1 0   0 0   0 1   1 1   1 1   1 0   0 0   0 0   0 1   1 0   0 1

0 0   0 1   1 0   0 1   1 1   1 0   0 0   0 1   1 1   1 1   1 0   0 0   0 0   0 1   1 0   0 1
0 1   1 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 0   0 1   1 0   0 1   1 1   1 0   0 0

0 1   1 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 0   0 1   1 0   0 1   1 1   1 0   0 0
1 1   1 0   0 0   0 0   0 *   * *   * 0   0 1   1 0   0 1   1 1   1 0   0 1   1 1   1 0   0 0

1 1   1 0   0 0   0 0   0 *   * *   * 0   0 1   1 0   0 1   1 1   1 0   0 1   1 1   1 0   0 0
0 0   0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 0

0 0   0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 0
0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 1   1 1   1 0   0 1

0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 0   0 0   0 0   0 1   1 1   1 0   0 1
0 0   0 1   1 0   0 1   1 1   1 1   1 0   0 0   0 1   1 0   0 *   * 1   1 1   1 0   0 0   0 0

0 0   0 1   1 0   0 1   1 1   1 1   1 0   0 0   0 1   1 0   0 *   * 1   1 1   1 0   0 0   0 0
0 0   0 1   1 0   0 1   1 0   0 0   0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 1

0 0   0 1   1 0   0 1   1 0   0 0   0 0   0 0   0 0   0 0   0 1   1 1   1 1   1 1   1 1   1 1
0 1   1 1   1 0   0 0   0 *   * 0   0 0   0 0   0 0   0 1   1 1   1 0   0 0   0 0   0 0   0 0

0 1   1 1   1 0   0 0   0 *   * 0   0 0   0 0   0 0   0 1   1 1   1 0   0 0   0 0   0 0   0 0
0 0   0 0   0 0   0 0   0 *   * *   * 0   0 0   0 0   0 0   0 0   0 0   0 0   0 0   0 0   0 0
```

`map5`

Diese Landschaft kann nicht ohne Regelverletzung vervollständigt werden.

```
1 1   1 1   1 1   1 1   1 1
1 0   0 0   0 0   0 1   1 1

1 0   0 0   0 0   0 1   1 1
0 0   0 1   1 0   * *   0 0

* *   0 1   * *   1 0   0 0
* *   0 0   * *   0 0   0 0

* *   0 0   0 1   * *   0 0
* *   1 1   1 1   * *   1 1

0 1   1 1   1 1   1 1   1 1
1 1   1 0   0 0   0 1   1 1
```

Quellcode

Die MapGenerator-Klasse

`__init__(fileobj)`

Der Konstruktor liest die Landschaft aus dem übergebenen Dateiobjekt ein und setzt die in der Lösungsidee beschriebenen Attribute.

`to_string(spacy=False, size=False, empty_color="\033[1;32m", not_found_color="\033[1;31m")`

Diese Methode erstellt die unter „Beispiele“ verwendete Ausgabe. Da die Funktion zur Lösung der Aufgabe nicht relevant ist, wird sie hier nicht weiter erklärt.

`fill_spaces(wildcard_fill=0)`

Diese Methode ergänzt die Landschaft. Bits, für die es mehrere Möglichkeiten gibt, werden auf `wildcard_fill` gesetzt.

```
for y, x in self.original_empty_parts:
    if self.parts[y, x] == 2:
        # Quadrat (y, x) wurde noch nicht gesetzt
        self._set_bit(y, x)
# alle übriggebliebenen Bits auf wildcard_fill setzen
for pos in self.empty_parts:
    assert self.parts[pos[0], pos[1]] == 2
    self.parts[pos[0], pos[1]] = wildcard_fill
```

`_set_bit(y, x)`

Diese Methode sucht die zwei benachbarten, relevanten Quadrate und prüft, ob der Bit für das Quadrat (y, x) – im Moment leer – gesetzt werden kann und tut dies ggf.

```
# benachbarte, relevante Quadrate ("surrounding") ermitteln
surrounding = []
if y % 2 == 0:
    # das Quadrat (y, x) befindet sich in der oberen Hälfte einer Kachel
    surrounding.append((y - 1, x))
else:
    # untere Hälfte
    surrounding.append(((y + 1, x)))
```

```
if x % 2 == 0:
    # linke Hälfte
    surrounding.append((y, x - 1))
else:
    # rechte Hälfte
    surrounding.append((y, x + 1))
possible_bit = None  # Bit für das Quadrat (y, x)
empty_surrounding = None  # benachbartes Quadrat, das leer ist (falls es existiert). Wird, falls
                          # ein Bit gefunden wird, als nächstes besucht (für 2. Fall).
for y_, x_ in surrounding:
    if 0 <= y_ < self.parts.shape[0] and 0 <= x_ < self.parts.shape[1]:
        # (y_, x_) befindet sich in der Landschaft. Alle Quadrate außerhalb der Landschaft
        # können als beliebig angesehen werden.
        bit = self.parts[y_, x_]
        if bit != 2:  # nicht leer
            if possible_bit is not None and bit != possible_bit:
                # possible_bit wurde bereits vom ersten benachbarten, relevanten Quadrat
                # gesetzt. Die Bits unterscheiden sich.
                self.no_bit_found_pos = (y, x)  # Für rote Markierung in der Ausgabe Position
                                                # des Quadrats speichern
                raise ValueError(f"Keine Kachel für ({y / 2}, {x / 2}) gefunden")
            possible_bit = bit
        else:
            # Wenn dies das erste der zwei benachbarten, relevanten Quadrate ist, könnte
            # empty_surrounding vom zweiten Quadrat - falls dieses auch leer ist - überschrieben
            # werden. Dann wäre aber kein Bit gefunden worden und empty_surrounding wird nicht
            # benutzt.
            empty_surrounding = (y_, x_)
if possible_bit is not None:
    self.parts[y, x] = possible_bit
    self.empty_parts.remove((y, x))
    if empty_surrounding:
        # Eines der beiden benachbarten Quadrate war leer, dieses wird nun besucht, da nun
        # entweder der Wert des Bits feststeht oder eine Regelverletzung festgestellt wird.
        self._set_bit(*empty_surrounding)
```