Aufgabe 1: Arukone

Team-ID: 00395

Team: DieInformatiker

Bearbeiter/-innen dieser Aufgabe:
Benedikt Wiesner, Leonard Beinlich

05. November 2023

Lösungsidee

Unser Ziel bestand darin, ein Programm zu entwickeln, das Arukone-Rätsel dynamisch generiert, wobei die erstellten Rätsel bestimmten Regeln entsprechen. Diese Regeln lauten:

- Gittergröße n x n: Das Programm sollte in der Lage sein, Rätsel in verschiedenen Gittergrößen zu erstellen, wobei das Gitter stets quadratisch ist.
- Die Anzahl der Paare: Die Anzahl der Paare soll mehr als die Hälfte der Größe des Feldes entsprechen.
- Zwei Vorkommen jeder Zahl: Jede Zahl im Gitter soll genau zweimal vorkommen, um das Rätsel lösbar zu machen
- Verbindung der Zahlen: Die generierten Rätsel sollen gemäß den Arukone-Regeln verbunden werden. Dabei dürfen die Verbindungen nur horizontal oder vertikal sein und müssen jedes Paar gleicher Zahlen mit den entsprechenden Zahlen miteinander verbinden.

Vorgehensweise

1. **Gittererstellung:** Unser Ansatz beginnt mit der Erstellung eines leeren Gitters der Größe n x n, das als Grundlage für das Arukone-Rätsel dient. Dies ist eine zweidimensionale Array-liste.
2. **Zufällige Platzierung der Zahlen:** Anschließend platziert das Programm die Zahlen gemäß den vorgegebenen Regeln auf dem Gitter. Dabei wird darauf geachtet, dass jede Zahl genau zweimal vorkommt und die Platzierung der Zahlen eine Lösung ermöglicht.
   Die Zahlen werden in einer zufälligen Reihenfolge platziert. Unser Programm platziert ein zufällige Zahlenpaar und verbindet dieses mit einer Linie. Anschließend wird das nächste Zahlenpaar platziert und auch direkt verbunden, um das Generieren eines lösbaren Arukone Rätzels zu garantieren.

3. **Verbindung der Zahlen:** Nachdem die Zahlen platziert wurden, werden auf direktem Weg miteinander verbunden, wobei jede Zahl exakt zwei Verbindungen hat, die nur horizontal oder vertikal verlaufen. Dies wird auch ausgegeben.

Unser Ansatz konzentrierte sich darauf, eine effiziente Möglichkeit zu finden, um Rätsel mit variabler Schwierigkeit zu generieren, wobei die Anforderungen an die Rätselstruktur erfüllt werden. Es ist wichtig anzumerken, dass einige der generierten Rätsel von unserem Programm nicht gelöst werden können, was eine zusätzliche Herausforderung für die Lösung dieser Rätsel darstellt.

Umsetzung

Unser Programm beginnt mit der Erstellung eines Gitters der Größe n * n, wobei n eine vom Benutzer eingegebene Zahl ist. Dieses Gitter besteht aus Feldern, die durch die Klasse Feld definiert werden. Die Platzierung der Zahlen im Rätsel erfolgt zufällig, wobei darauf geachtet wird, dass jede Zahl genau zweimal vorkommt und mindestens die Hälfte der Zahlen im Rätsel enthalten sind (Das Programm prüft nach gültig oder ungültig). Um den kürzesten Weg zu finden haben wir einen Breitensuchalgorithmus (BSF) verwendet. Der Breitensuchalgorithmus im Allgemeinen beginnt an einem Startknoten und arbeitet schichtweise durch den Graphen, indem er benachbarte Knoten durch eine Warteschlange besucht. Dadurch findet er alle erreichbaren Knoten vom Startpunkt aus. Am Ende wird die Lösung als txt Datei ausgegeben, um sie in das Challenger-Programm einfügen zu können.
Hier sind die Kernaspekte der Umsetzung:

Klassenstruktur

Feld-Klasse:
Die Implementierung umfasst die Definition einer Klasse namens Feld, die Informationen über die Felder im Gitter speichert. Diese Klasse hält Eigenschaften wie Koordinaten, Wert, Belegung und Freiheit eines Feldes.

Funktionen und Methoden

Gittererstellung:

- Die Funktion `felderstellen_feld()` wird genutzt, um ein zweidimensionales Array zu erstellen, das als Gitter für das Arukone-Rätsel dient. Dabei werden alle Felder initialisiert und für die weitere Verwendung vorbereitet.

Zufällige Platzierung der Zahlen:

- Die Funktion `zahleneinfuegen()` platziert Zahlen gemäß den Regeln des Rätsels auf dem Gitter, indem sie zufällige Positionen auswählt und die Zahlen entsprechend platziert.

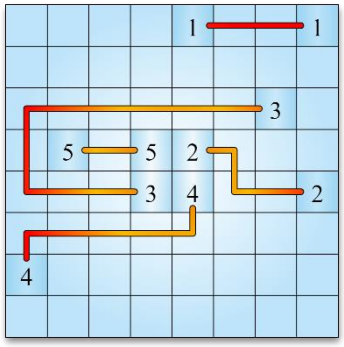
Verbindung der Zahlen:

- Die Methode `ereichbareFelder(zahl)` sucht nach freien Feldern, die von einer gegebenen Zahl aus erreicht werden können, und markiert diese als belegt, um die Verbindung zwischen den Zahlen zu ermöglichen.
- Die Methode `kuerzestenWegBekommen()` findet mit Hilfe eines Breitensuchalgorithmus den kürzesten weg und verbindet die paare dementsprechend.

Beispiele

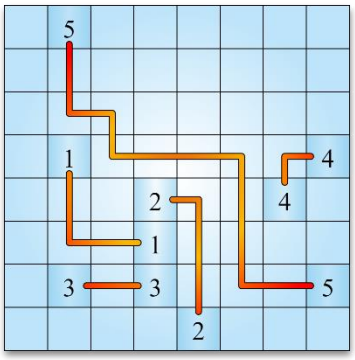
1.

```
Gib eine Zahl ein: 8
Gueltig
Gib die Menge der Zahlenpare an diese muessen mehr als die Haelfte der Groesse des Feldes entsprechen:  5
Gueltig
8
5
0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 3 0
0 5 0 5 2 0 0 0
0 0 0 3 4 0 0 2
0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0
3 3 3 3 3 3 3 0
3 5 5 5 2 2 2 2
3 3 3 3 4 0 0 2
0 0 0 0 4 0 0 0
4 4 4 4 4 0 0 0
0 0 0 0 0 0 0 0
```



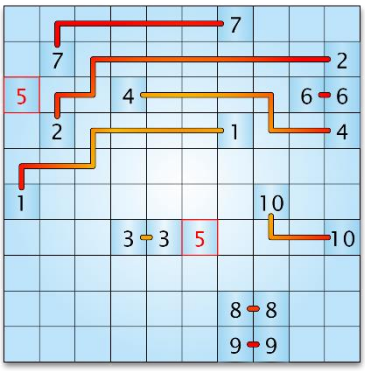
2.

```
Gib eine Zahl ein: 8
Gueltig
Gib die Menge der Zahlenpare an diese muessen mehr als die Haelfte der Groesse des Feldes entsprechen:  5
Gueltig
8
5
0 5 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 4
0 0 0 2 0 0 4 0
0 0 0 1 0 0 0 0
0 3 0 3 0 0 0 5
0 0 0 0 2 0 0 0

0 5 0 0 0 0 0 0
0 5 0 0 0 0 0 0
0 5 5 0 0 0 0 0
0 1 5 5 5 5 0 4
0 1 0 2 2 5 4 4
0 1 1 1 2 5 0 0
0 3 3 3 2 5 5 5
0 0 0 0 2 0 0 0
```



3.

```
10Gib eine Zahl ein:
Gueltig
Gib die Menge der Zahlenpare an diese muessen mehr als die Haelfte der Groesse des Feldes entsprechen:  10
Gueltig
10
10
0 0 0 0 0 0 7 0 0 0
0 7 0 0 0 0 0 0 0 2
5 0 0 4 0 0 0 0 6 6
0 2 0 0 0 0 1 0 0 4
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 10 0 0
0 0 0 3 3 5 0 0 0 10
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 8 8 0 0
0 0 0 0 0 0 9 9 0 0

0 7 7 7 7 7 7 0 0 0
0 7 2 2 2 2 2 2 2 2
5 2 2 4 4 4 4 4 6 6
5 2 0 0 0 0 1 4 4 4
5 5 5 5 5 5 1 0 0 0
1 0 0 0 0 5 1 10 0 0
1 0 0 3 3 5 1 10 10 10
1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 8 8 0 0
0 0 0 0 0 0 9 9 0 0
```



Quellcode

Eingabe und Überprüfung der Gültigkeit der eingegebenen Zahl:

```
n = int(input("Gib eine Zahl ein: "))

if n <= 4:
    print("Ungueltig")
else:
    print("Gueltig")

p = int(input("Gib die Menge der Zahlenpare an diese muessen mehr als die Ha-
elfte der Groesse des Feldes entsprechen:  "))

if p >= ((n/2)+1+int(n%2)):
    print("Gueltig")
else:
    print("Ungueltig")
```

Klasse zur Verwaltung von Feldinformationen:

```
class Feld():
    def __init__(self):
        self.x_kordinnate = int
        self.y_kordinnate = int
        self.belegt = None
        self.wert = int
        self.frei = None
```

Erstellung des Rätselgitters:

```
def felderstellen_feld():
    zweid_array = []
    for i in range (n):
        zeile = []
        for j in range (n):
            feld = Feld()
            feld.x_kordinnate = j
            feld.y_kordinnate = i
            feld.wert = 0
            feld.belegt = False
            feld.frei = True
            zeile.append(feld)

        zweid_array.append(zeile)
    return zweid_array
```

Funktion zum Einsetzen von Zahlen an zufälligen Stellen:

```
def zahleneinfuegen():
    i = 1
    while i <= (p):
        x = randomX()
        y = randomY()
        if zweideminsionale_liste[y][x].frei == True:
            zweideminsionale_liste[y][x].wert = i
            zweideminsionale_liste[y][x].frei = False
            i = i+1
    return zweideminsionale_liste
```

Hauptteil des Programms:

```
n = int(input("Gib eine Zahl ein: "))  # Benutzereingabe für die Größe des
Spielfelds
zweideminsionale_liste = felderstellen_feld()  # erstellt das Spielfeld
zahleneinfuegen()  # fügt Zahlen an zufälligen Stellen ein
```

Erhalten von möglichen Bewegungskoordinaten:

```
def naechsteBewegungBekommen(x, y):
    return {
        'left':  [x-1, y],
        'right': [x+1, y],
        'up':  [x, y-1],
        'down':  [x, y+1]
    }.values()
```

Breitensuchalgorithmus:

```
def kuerzestenWegBekommen(level, startKoordinate, endKoordinate):
    wegFinden = [[startKoordinate]] # Liste, um Wege zu finden
    besuchteKoordinaten = [startKoordinate] # Besuchte Koordinaten

    while wegFinden != []:
        aktuellerWeg = wegFinden.pop(0) # Aktueller Weg
        aktuelleKoordinate = aktuellerWeg[-1] # Aktuelle Koordinate

        currentX, currentY = aktuelleKoordinate

        if aktuelleKoordinate == endKoordinate: # Wenn Endkoordinate erreicht ist
            return aktuellerWeg

        for nextKoordinate in naechsteBewegungBekommen(currentX, currentY):
            nextX, nextY = nextKoordinate

            # Überprüfung der Grenzen des Levels

            if nextX < 0 or nextX >= levelWidth:
                continue

            if nextY < 0 or nextY >= levelHeight:
                continue
            # Wenn Koordinate bereits besucht wurde oder Feld nicht frei ist
```

```
            if nextKoordinate in besuchteKoordinaten:
                continue

            if level[nextY][nextX].frei == False:
                continue

            # Hinzufügen des aktuellen Wegs mit der nächsten Koordinate
            wegFinden.append(aktuellerWeg + [nextKoordinate])
            # Markieren der nächsten Koordinate als besucht
            besuchteKoordinaten += [nextKoordinate]
```