Aufgabe 1: Arukone

Team-ID: 00776

Team: BWINF0

Bearbeiter/-innen dieser Aufgabe:
Tobias Maurer

19. November 2023

Inhaltsverzeichnis

Lösungsidee

Der von BWINF zur Verfügung gestellte Arukonelöser ist ein sehr simples Programm. Als ich mit dem ein wenig experimentiert habe, entgingen diesem teils simple Lösungen. Daher muss man nur ein Arukonerätsel generieren, das in der Komplexität Arukonefelder generiert, die komplex genug sind, damit der gegeben Aurkonelöser keine Lösung mehr findet.

Umsetzung

Das Arukonefeld kann man durch eine 2-dimensionale Liste mit den entsprechenden Größen dargestellt werden. Als nächstes benötigt man alle noch freien Felder und eine Möglichkeit alle erreichbaren Felder zu von einer gegebenen Startposition aus zu berechnen. Das Programm verwendet auch $\text{size\_per\_ration} = \lceil \frac{n^2}{2 \cdot pairs} \rceil$. Diese Formel ergibt 1, wenn jedes Feld belegt werden müsste, um alle Paare zu integrieren. Genauer gesagt, ist es in etwa die Durchschnittslänge der Wege um Paare miteinander zu verbinden. Wenn hierbei eine Zahl <=2 herauskommt, wird mein Programm vermutlich kein Arukonerätsel finden, dass die entsprechende Paare in die entsprechende Feldgröße einbaut. Es wird dennoch versucht, aber die Wahrscheinlichkeit ist zu gering, da es auf Zufall basiert. Ich habe mich nicht weiter darum gekümmert, da das resultierende Arukone sowieso keinen Spaß zu lösen machen würde und auch die Komplexität wäre sehr gering, da nahezu alle Paare benachbart wären: Der Arukonelöser könnte das Ergebnis lösen, aber mein Programm soll Arukone lösen, die nicht von dem gegebenen Arukonelöser lösbar sind.

Es werden am Ende mindestens $\lceil \frac{n}{2} \rceil$ Paare berechnet. Jedes Paar wird mit der folgenden Abfolge berechnet:

1. Es wird eine zufällige noch freie Position gewählt. Diese Position wird als Startpunkt verwendet
2. Alle erreichbaren Felder von dieser Position werden berechnet. Diese Positionen werden absteigend der Länge (die Anzahl der Felder für die Verbindung) nach sortiert.
3. Danach werden nur die ersten (*size_pair_ratio* – 1) Felder beibehalten. Davon werden ebenfalls, falls möglich (mindestens 1 muss noch da bleiben), alle Felder, deren Entfernung kleiner *size_pair_ratio* entfernt. Dies garantiert möglichst lange Wege, aber gleichzeitig wird noch der Zufallsaspekt beibehalten, wodurch der Algorithmus nicht mehr deterministisch ist.
4. Alle Felder, die man als Verbindung benötigt, werden mit der entsprechenden Paarnummer in der 2-dimensionalen Liste markiert, damit die Wegsuche weiß, dass sie dort nicht entlang kann.
5. Alle Felder, die für den Weg sowie Start- und Endpunkt, werden aus den noch freien Positionen entfernt.

Es kann passieren, dass man kein Feld erreichen kann, welches noch verfügbar war. In diesem Falle wird es einfach entfernt und es wird geschaut, ob es überhaupt noch freie Felder gibt. Im Falle, dass das nicht so ist, wird der Lösungsalgorithmus rekursiv erneut gestartet (Limit liegt bei 100 Rekursionen). Ansonsten wird für dasselbe Paar wieder bei Schritt 1 von oben begonnen. Sofern das Paar erfolgreich erstellt wurde, wird dieses in ein Mapping, welches die Paarnummer und Start- und Ende enthält, geschrieben für die spätere Ausgabe, da im ursprünglichen Feld auch der Lösungsweg steht.

Die Paarnummern werden rückwärts vergeben. D. h. bei 5 Paaren, wird als Erstes Paar 5 erstellt, danach folgt 4, etc. Dies hat sich als gute Strategie gezeigt, da längere Strecken automatisch die Komplexität erhöhen. Oft haben auch die kleineren Paarnummern, die kürzeren Strecken. Dies macht auch Sinn, da am Ende das Feld viel mehr ausgefüllt ist als am Anfang.

Die Feldsuche funktioniert durch das rekursive Besuchen von freien Nachbarfeldern. Ein Nachbarfeld zählt dann als frei, wenn an dieser Stelle kein Paar dieses bisher als Weg-, Start- oder Endpunkt verwendet (es den Wert 0 hat). Es beginnt die Nachbarn vom Startfeld zu besuchen. Danach die Nachbarn von den Nachbarn des Starts usw. Die Pfade werden ebenfalls gespeichert für einerseits das Markieren des Pfades nach dem Wählen des erreichbaren Felds, aber andererseits ist es auch nicht erwünscht, dass das ganze Feld direkt von einem Paar gefüllt ist und somit keinen Raum mehr für weitere Paare gefunden wird. Deshalb wird im Falle von einem bereits besuchten Feld geschaut, ob der neue Pfad kürzer ist und falls so sei, wird das aktualisiert. Von diesem Feld wird dann auch nochmal neu berechnet, welche Felder erreichbar sind, da möglicherweise kürzere Wege existieren. Es ist zwar widersprüchlich nach kurzen Wegen zu suchen, wenn man lange Wege finden möchte. Allerdings will man den **kürzesten** Weg zum **weitentfertesten** Feld vom Startfeld

aus. Zudem werden die Wege automatisch komplexer je weniger Möglichkeiten es gibt über die Felder zu „wandern“.

Am Ende wird das ungelöste Arukon ausgegeben in validem Format, damit man es in den Arukonelöser kopieren kann und das gelöste Arukon mit den Wegen, welches am Ende sowieso in der 2-dimensionalen Liste steht.

Programmhinweise

Ausgabe

Bei der Ausgabe kann es passieren, dass während der Berechnung Arukonerätsel und Lösungen ausgegeben werden, sowie INFO, FAILURE, … Nachrichten. Diese Arukone sind sozusagen Zwischenergebnisse, die nicht die erforderlichen Parameter erfüllen, aber vermutlich nahe dran sind. Um die anderen Nachrichten braucht man sich eigentlich nicht zu kümmern. Wenn das Programm "FAILURE: Failed finding a solution within 100 tries” ausgibt, hat es die rekursiven Aufrufe, um das Arukon zu generieren aufgebraucht. Dies passiert vor allem, wenn man besonders viele Paare in das Feld integrieren will, da das Programm am Anfang möglichst lange Wege wählt, was es in diesem Fall nicht machen sollte. Aber, wie bereits gesagt, ist das Ergebnis höchstwahrscheinlich nicht wünschenswert, weshalb ich es so gelassen habe, um bei erwünschten Ergebnissen ein gutes Ergebnis zu erzielen.

Ausführung

Das Programm setzt das Rekursionslimit auf 5000.

Im Falle, dass das Programm keine Terminalargumente bekommt, wird man einfach nach der Feldgröße und den Paaren gefragt.

Wenn das Programm Terminalargumente bekommt, werden diese abwechselnd als Feldgröße und Anzahl der Paare gelesen:

```
python3 arukone.py 6 5 10 8
```

Obiger Befehl wird vom Programm verstanden als:

Das erste Arukon mit n=6 und 5 Paaren und noch ein zweites Arukon mit n=10 und 8 Paaren

Beispiele

Alle hier aufgeführten Beispiele befinden sich ebenfalls im Ordner “results” dieser Aufgabe sowie weitere Beispiele die von meinem Programm generiert wurden

arukone1.txt

6

5

0 0 4 0 0 0

1 0 0 1 0 3

0 0 0 5 0 0

0 0 0 0 0 0

0 4 0 0 0 0

5 0 0 3 2 2

An diesem Beispiel kann man sehen, dass am Ende (also die kleinen Paarnummern) nur noch kurze Wege gefunden werden und eher am Anfang lange Wege entstehen. Der Weg der 4 ist allerdings der komplexeste und dies ist an dem vorher generierten Paar (die 5) geschuldet. Dadurch ging der direkte Weg nicht und der Weg für die 4 musste sich außen drum bauen. Das die 5 den Weg der 4 blockiert sieht man in der zusätzlich ausgegebenen Lösung von meinem Programm:

0 0 4 4 4 0

1 1 1 1 4 3

5 5 5 5 4 3

5 4 4 4 4 3

5 4 0 3 3 3

5 0 0 3 2 2

Auch wenn das Arukonrätsel nicht von dem BWINF Löseprogramm gelöst werden kann, ist das leicht umzusetzeh, in dem man die unten aufgeführte Änderung vornimmt.

Änderung von Hand (3 und 4 getauscht):

6

5

0 0 3 0 0 0

1 0 0 1 0 4

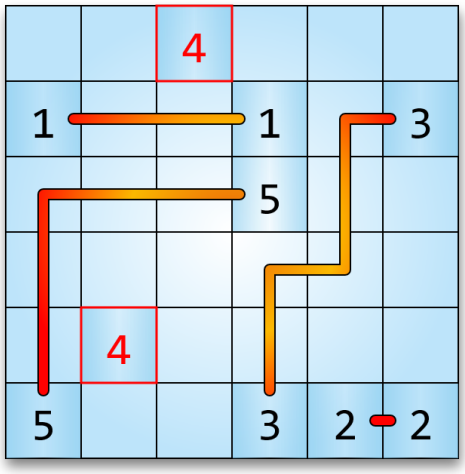
0 0 0 5 0 0

0 0 0 0 0 0

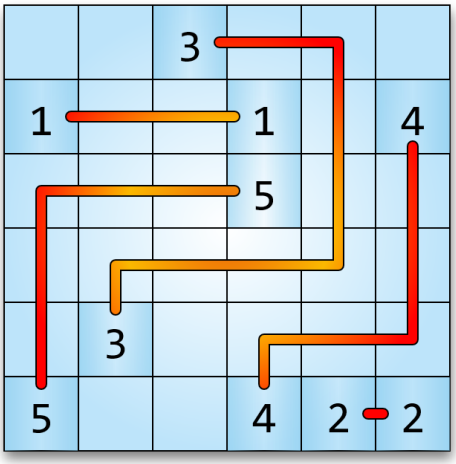
0 3 0 0 0 0

5 0 0 4 2 2

Die Ausgabe bei dem nicht gelösten Arukon oben:



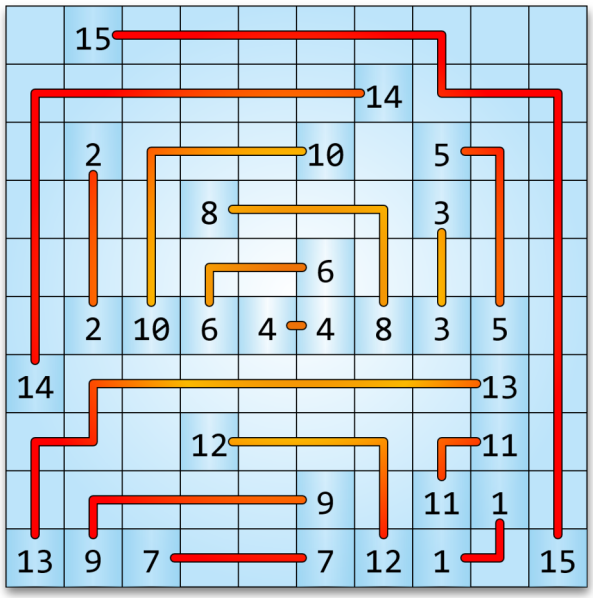
Die Ausgabe mit der oben gezeigten Änderung:



Meine Annahme ist, dass die Paare von 1 beginnend ihrer Nummerierung nach gelöst werden. Die 1 verhindert den direkten Weg von oben, weshalb sich die 3 dann herum schlängelt und die 4 am Ende das Gegenstück findet. Im generierten Arukon wird die 3 (also im Bild auf dieser Seite die 4) allerdings zuerst generiert, wodurch der gewählte Weg blockiert wird.

arukone6.txt

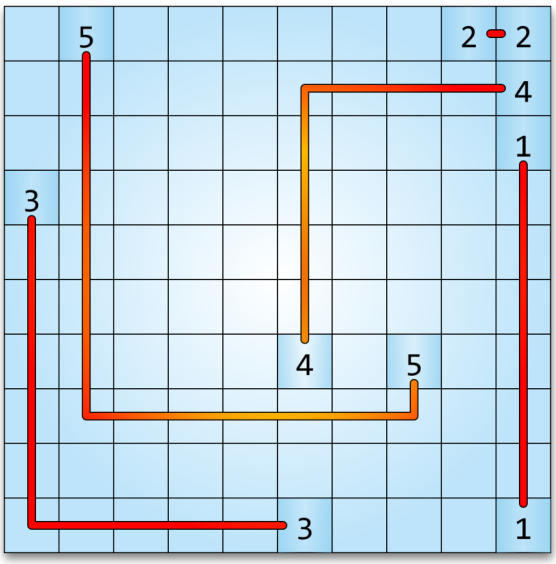
Die Textdarstellung befindet sich in Aufgabenordner in “results/arukone6.txt” aufgrund des Textumfangs.



Bei diesem Arukon sieht man, dass die hohen Zahlen weiterhin eher lange Wege im Vergleich zu den kleineren Zahlen haben. Da allerdings viele Paare in das Arukon gepackt werden sollten, sind die Zahlen recht dicht beieinander am Ende, wodurch die Komplexität recht gering ist. Es kommt also auch darauf an nicht zu viele Paare in ein zu kleines Feld zu packen.

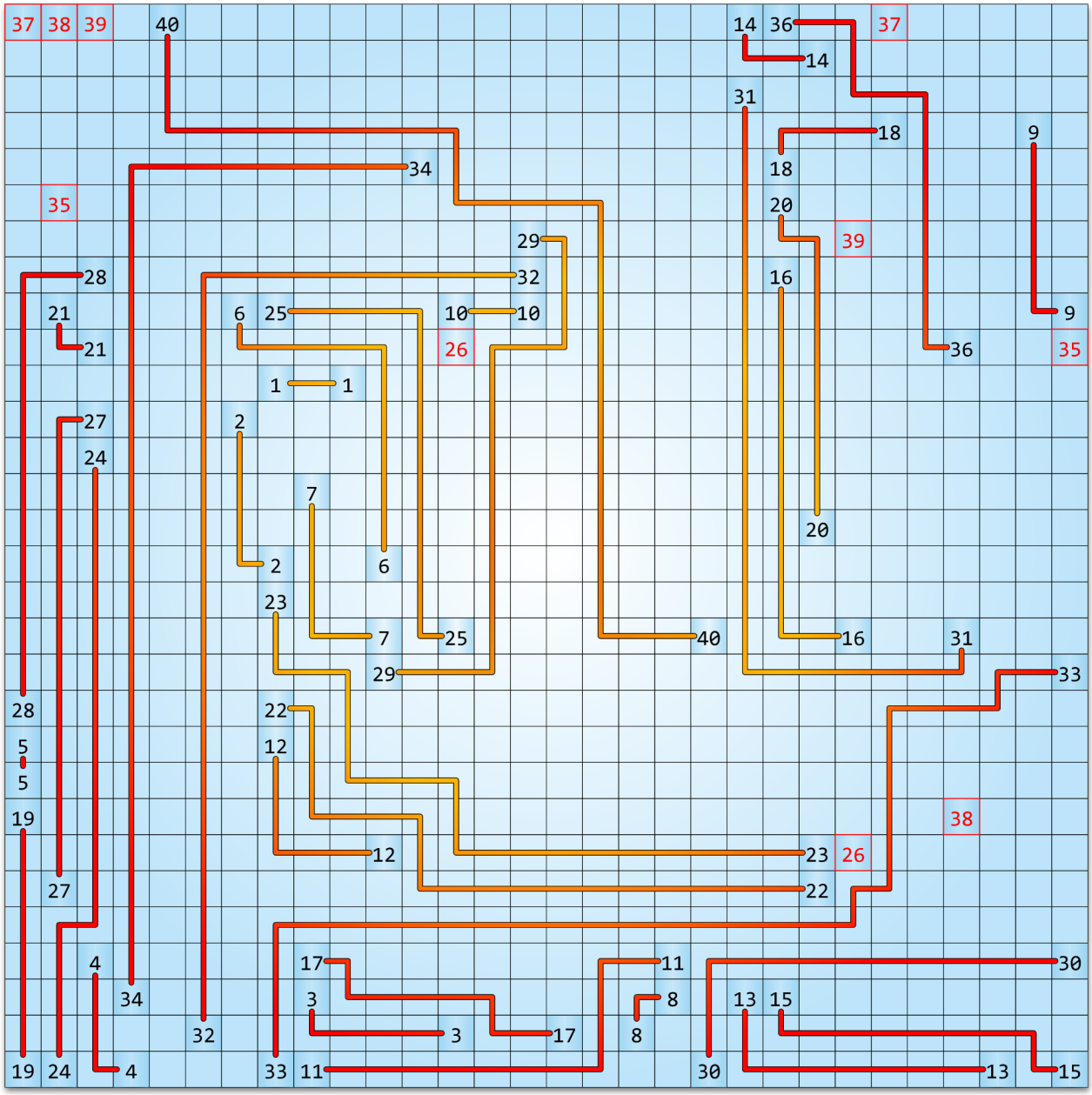
arukone7.txt

Textdarstellung befindet sich wie davor auch in der entsprechenden Datei “results/arukone7.txt“



Nach dem vorherigen Problem mit zu viel Paaren meint man, man könnte einfach sehr wenige nehmen. Allerdings sind zu wenig Paare auch wieder ein Problem, da die Wege sich sehr wenig kreuzen und somit der Arukonelöser wenig Behinderung haben wird.

arukone8.txt enthält ein Arukon mit 40 Paaren und einer Feldgröße von 30x30 (das Maximum, welches erlaubt ist)



Quellcode

Verwendung von externen Programmen: math (ceil), random (choice (wählen eines zufälligen Elements einer Liste), randint), sys (recursionlimit), time (1-second waiting, nach Terminaloutput), typing (für typehints), attr (dataclasses, um schnell ein Klasse mit nur 2 Attributen zu haben, um ein Feld (Point) zu repräsentieren)

Berechnung der erreichbaren Felder

```
34  def get_neighbours(pos: Point, field) -> Generator[Point, None, None]:
35      for dy in range(-1, 2, 1):
36          for dx in range(-1, 2, 1):
37              if pos.x + dx < 0 or pos.x + dx >= len(field) or pos.y + dy < 0 or pos.y + dy >= len(field) or abs(dx) + abs(dy) == 2:
38                  continue
39
40              n = Point(pos.x + dx, pos.y + dy)
41              if field[n.y][n.x] == 0:
42                  yield n
43
44
45  def get_reachable_positions(start: Point, field: List[List[int]], data: Dict[Point, List[Point]]) -> Dict[Point, List[Point]]:
46      if not data:
47          data[start] = [start]
48      for neighbour in get_neighbours(start, field):
49          if data.get(neighbour):
50              if len(data[start]) + 1 >= len(data[neighbour]):
51                  continue
52          data[neighbour] = data[start][:] + [neighbour]
53          data.update(get_reachable_positions(neighbour, field, data))
54      return data
```

Initialisierung der Variablen

```
70          field = [[0 for _ in range(n)] for _ in range(n)]
71
72          positions: Dict[int, Tuple[Point, Point]] = {}
73          free_positions = [Point(x, y) for y in range(0, n) for x in range(0, n)]
74          size pair ratio = math.ceil(n * n / (2 * pairs))
```

nächste Seite

Die Schleife, wo alle Paare generiert werden:

```python
def solve(n: int, pairs: int, depth=100):
    while len(positions) < pairs:
        if not free_positions:
            print("Failure: No free positions anymore.\nCurrent result:")
            print_result(n, len(positions), positions, field)  # len(positions) for pairs to have the correct amount
            print("INFO: Next try")
            solve(n, pairs, depth-1)
            return

        # choosing start point
        pos1 = random.choice(free_positions)

        # finding every reachable field
        reachable_positions: Dict[Point, List[Point]] = get_reachable_positions(pos1, field, {})
        reachable_positions.pop(pos1)   # only help for the funtion (added in the function)
        if not reachable_positions:
            # single closed field with no neighbours
            free_positions.remove(pos1)
            if not free_positions:  # may happen in small fields and large amount of pairs
                print("FAILURE: Creating arukon. Retrying...")
                solve(n, pairs, depth-1)
                return
            continue

        # larger distances = more difficult to solve
        lpositions = sorted(list(reachable_positions), key=lambda x: len(reachable_positions[x]), reverse=True)
        if len(lpositions) > 5:
            lpositions = lpositions[:size_pair_ratio]

        # remove distances of size_pair_ratio or lower if possible  (n=6; pairs=5) = ceil(36 / 10) = 4  (distances of 4 or lower)
        for i in range(len(lpositions) - 1, -1, -1):
            if len(lpositions) == 1:
                break
            elif len(reachable_positions[lpositions[i]]) - 1 <= size_pair_ratio:
                lpositions.pop(i)

        # choosing end point
        index = random.randint(0, len(lpositions) - 1)
        pos2 = lpositions[index]

        # marking path
        for p in reachable_positions[pos2]:
            field[p.y][p.x] = pairs - len(positions)
            free_positions.remove(p)
        positions[pairs - len(positions)] = (pos1, pos2)

    # outputting solution and unsolved arukon
    print_result(n, pairs, positions, field)
```