

# Aufgabe 1: Arukone

Team-ID: 00585

Team-Name: Unendlich Pfanddosen

Bearbeiter dieser Aufgabe: Robert Vetter

10. November 2023

## Inhaltsverzeichnis

<b>1 Aufgabe 1</b>	<b>1</b>
1.1 Lösungsidee . . . . .	1
1.2 Umsetzung . . . . .	2
1.3 Beispiele . . . . .	4
1.4 Anhang . . . . .	7
1.4.1 Quellcode . . . . .	7

# 1 Aufgabe 1

## 1.1 Lösungsidee

Die Herausforderung liegt in der Erstellung von Arukone-Rätseln, die bestimmte Bedingungen erfüllen und eine gewisse Komplexität aufweisen, um nicht von dem einfachen vom Bundeswettbewerb Informatik zur Verfügung gestellten Programm gelöst werden zu können. Dabei darf ein Linienzug nur aus horizontalen und vertikalen Abschnitten bestehen, wodurch schräg verlaufende Pfade ausgeschlossen sind. Des Weiteren beginnt oder endet in jedem Feld mit einer Zahl ein Linienzug und jedes Feld ohne Zahl wird von genau einem Linienzug durchlaufen oder ist leer.

Meine Lösungsidee basiert auf der zufälligen Auswahl von Start- und Endpunkten, welche die Eckpunkte der Linienzüge repräsentieren. Diese Punkte sind die Zahlen im Rätsel, die durch Linienzüge verbunden werden sollen und noch nicht benutzt wurden. Zur Suche eines möglichen Pfades zwischen zwei Punkten gibt es verschiedene Algorithmen, die angewendet werden könnten, wie beispielsweise die Tiefensuche (DFS), Breitensuche (BFS), Backtracking und den A\*-Algorithmus. Dabei entschied ich mich für Letzteren, da er durch den Einsatz von Heuristiken zielgerichtet nach einem möglichst kurzen Weg suchen kann. Da es außerdem nicht gefordert ist, den kürzesten Weg zwischen beiden Punkten zu finden, bietet sich eine Heuristik an. Die Heuristik garantiert nicht den optimalen Weg, ermöglicht jedoch eine sehr effiziente Lösungssuche mit geringer Zeitkomplexität.

Sobald ein Pfad gefunden wurde, wird der Prozess mit dem Generieren zweier weiterer unbenutzter zufälliger Start- und Endpunkte fortgesetzt, zwischen denen dann wiederum nach einem Pfad gesucht wird. Dieser Prozess wird wiederholt, bis mindestens die Hälfte von  $n$  Pfaden ermittelt wurden. Dabei wird die Hälfte aufgerundet, wenn  $n$  ungerade ist. Darauf hinaus traf ich die Festlegung, dass ein Pfad mindestens eine Länge von drei Zellen besitzen muss, um als Pfad anerkannt zu werden. Dadurch werden zu kurze Linienzüge vermieden.

---

### Algorithm 1 Grundidee des Algorithmus

---

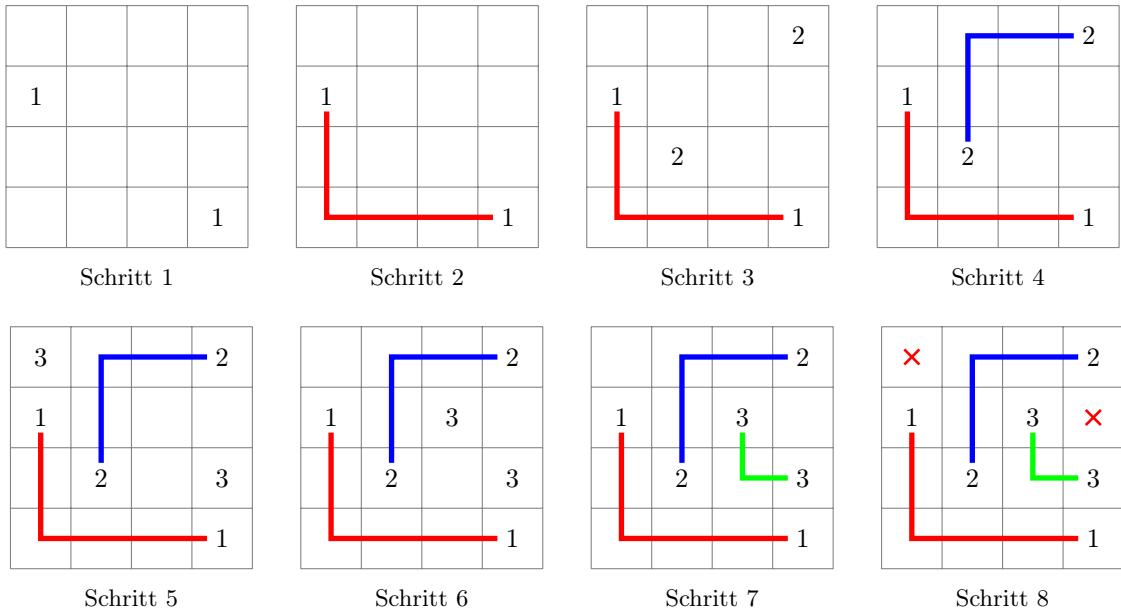
```

1: num_paths  $\leftarrow 0$ 
2: limit  $\leftarrow$  Hälfte von  $N$  (aufrunden, wenn  $N$  ungerade ist)
3: while num_paths  $< \text{limit}$  do
4:   grid  $\leftarrow$  initialize_grid( $N$ )
5:   while es gibt noch freie Zellen im Gitter do
6:     start, end  $\leftarrow$  wähle zufällige freie Zellen
7:     path  $\leftarrow$  finde Pfad zwischen start und end
8:     if Pfad ist gültig then
9:       füge Pfad zum Gitter hinzu
10:      num_paths  $\leftarrow$  num_paths + 1

```

---

Nun möchte ich die Arbeitsweise des Algorithmus beispielhaft an einem 4x4 Gitter erläutern. In der folgenden Abbildung werden im Schritt 1 zunächst zwei zufällige Punkte ausgewählt. Nun wird probiert, zwischen diesen zwei Punkten einen Pfad zu finden, der mindestens eine Länge von drei Zellen besitzt. Im Schritt 2 wurde dieser exemplarisch eingezeichnet. Entsprechend werden im Schritt 3 wiederum zwei Punkte ausgewählt und im Schritt 4 miteinander verbunden. Dieses wird im Schritt 5 wiederholt. Es ist jedoch unschwer erkennbar, dass nun kein möglicher Pfad existiert, ohne einen anderen Pfad zu übertreten. Dadurch werden zwei neue Punkte ausgewählt. Da zwischen ihnen eine Verbindung mit mindestens drei Zellen existiert, werden sie im Schritt 7 verbunden. Im letzten Durchlauf würde somit lediglich versucht werden, die beiden noch freien Zellen zu verbinden. Da dies nicht möglich ist, wird das erstellte Rätsel mit  $3 \geq 2$  minimalen Pfaden ausgegeben.



## 1.2 Umsetzung

Zur Umsetzung wählte ich als Programmiersprache Python, da sie sehr benutzerfreundlich ist und mit vielen verschiedenen möglichen Bibliotheken die Implementierung von Algorithmen erleichtert. Um meine Lösungsidee umzusetzen, schrieb ich zuerst eine Methode `initialize_grid` mit  $N$  als Parameter, die ein zweidimensionales Array, gefüllt mit Nullen, zurückgibt. Anschließend war es notwendig, eine Methode `get_neighbours` zu erstellen, welche alle Nachbarn einer Zelle zurückgibt, die innerhalb des Gitters liegen. Nachfolgend habe ich dies an zwei Beispielen erläutert. Angenommen, es sollen die Nachbarn der Zelle mit den Koordinaten  $(2, 1)$  ermittelt werden. Dann wären dies die Zellen mit den Koordinaten  $(1, 1)$ ,  $(2, 2)$ ,  $(3, 1)$  und  $(2, 0)$ . Im Beispiel 2 würden für die Ausgangszelle  $(0, 1)$  lediglich die Nachbarzellen  $(0, 0)$ ,  $(0, 2)$  sowie  $(1, 1)$  zurückgegeben werden.

			$(2, 0)$	
	$(1, 1)$	$(2, 1)$	$(3, 1)$	
		$(2, 2)$		

Beispiel 1

$(0, 0)$			
$(0, 1)$	$(1, 1)$		
$(0, 2)$			

Beispiel 2

Mithilfe von `get_neighbours` konnte ich nun eine Methode `get_empty_neighbours` schreiben, welche die Anzahl an noch freien Nachbarn zurückgibt.

Nun möchte ich mich dem Herzstück des Programms widmen. Der A\*-Algorithmus nimmt ein Gitter, einen Startpunkt, einen Endpunkt sowie die Größe  $N$  des Gitters als Parameter entgegen. Seine Funktionsweise basiert auf der Betrachtung der Nachbarn der aktuellen Zelle und dann auf der Auswahl des "besten" Nachbarn, um den Pfad fortzusetzen. Dieser Auswahlprozess stützt sich auf eine Bewertungsfunktion, die sowohl die bisherigen Kosten zum Erreichen der aktueller Zelle als auch die geschätzten zukünftigen Kosten zum Erreichen der Zielzelle (Heuristik) einbezieht. In diesem Fall wird die Manhattan-Distanz als Heuristik verwendet. Diese Distanz ist die Summe der absoluten Unterschiede der horizontalen und der vertikalen Koordinaten zwischen dem aktuellen Feld und dem Zielfeld:

$$\text{Heuristik}(a, b) = |a_x - b_x| + |a_y - b_y|$$

Hierbei sind  $a_x$  und  $a_y$  die x- und y-Koordinaten des Punktes  $a$ , und  $b_x$  und  $b_y$  sind die x- und y-Koordinaten des Punktes  $b$ .

Die Nachbarzelle, welche die niedrigste Bewertung in Bezug auf die kombinierten bisherigen und geschätzten zukünftigen Kosten hat, wird dann zur aktuellen Zelle. Der Algorithmus wiederholt diesen Prozess, indem er die Nachbarn des neuen aktuellen Feldes betrachtet und den besten auswählt, bis das Ziel erreicht ist oder bis festgestellt wird, dass kein Pfad zum Ziel existiert. Wenn das Ziel erreicht wird, wird der Pfad von Start zu Ziel rekonstruiert und zurückgegeben. Nachfolgend habe ich die grobe Funktionsweise des Algorithmus im Pseudocode dargestellt.

---

**Algorithm 2** A\*-Algorithmus

---

```

1: function ASTAR(grid, start, end, N)
2:   Node  $\leftarrow$  namedtuple("Node", ["total", "cell", "cost", "heuristic"])
3:   visited  $\leftarrow$  Set()
4:   queue  $\leftarrow$  Prioritätswarteschlange mit dem Startknoten
5:   came_from  $\leftarrow$  Dictionary()
6:   while queue  $\neq \emptyset$  do
7:     current  $\leftarrow$  heappop()
8:     if current.cell  $=$  end then
9:       return Rekonstruiere Pfad von came_from und gebe ihn zurück
10:    visited.add(current.cell)
11:    for each neighbour in Nachbarn von current.cell do
12:      if neighbour  $\in$  visited or grid[neighbour]  $\neq 0$  then
13:        continue
14:      new_node  $\leftarrow$  Node(neue Gesamtkosten, neighbour, neue Kosten, neue Heuristik)
15:      if neighbour  $\notin$  queue then
16:        came_from[neighbour]  $\leftarrow$  current.cell
17:        heappush(new_node)
18:    return []

```

---

Jetzt möchte ich noch einmal kurz auf die verwendeten Datenstrukturen eingehen.

- **Node**

Die Datenstruktur *Node* ist eine Instanz der `namedtuple`-Klasse in Python, die eine Zelle im Gitter (*cell*), die Kosten, um diese Zelle vom Startpunkt aus zu erreichen (*cost*), eine heuristische Schätzung der Entfernung zum Endpunkt (*heuristic*) und die Gesamtkosten (*total*) enthält. Sie merkt sich sozusagen die wesentlichen Informationen für jeden besuchten Knoten im Gitter, um den optimalen Pfad effizient zu berechnen und rekonstruieren zu können.

- **Visited**

Das Set *visited* speichert alle bereits besuchten Zellen, um sicherzustellen, dass der Algorithmus nicht im Kreis läuft.

- **Queue**

Diese Warteschlange speichert alle zu untersuchenden Zellen (alle freien Nachbarn der schon besuchten Zellen), wobei das Feld mit den geringsten Gesamtkosten (Kosten + Heuristik) zuerst entnommen wird. Dieses geschieht durch den Aufruf von `heappop`. Da `heappush` die Zellen basierend auf ihren Gesamtkosten einfügt und die Zellen mit den geringsten Gesamtkosten vorne stehen, ist diese Struktur mit einer Prioritätswarteschlange gleichzusetzen.

- **Came\_From**

Das Dictionary *Came\_From* speichert für jede Zelle die Vorgängerzelle auf dem besten bekannten Pfad vom Startpunkt aus.

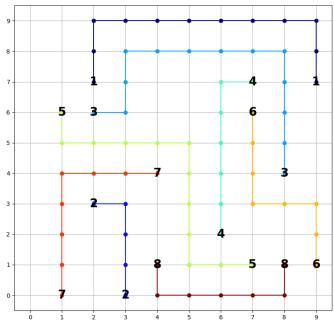
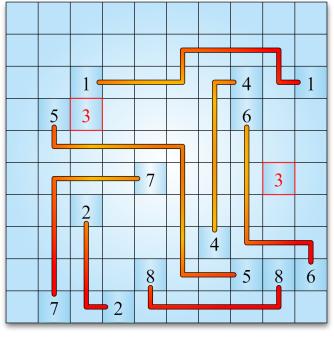
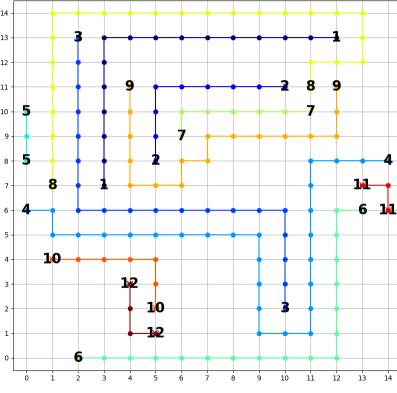
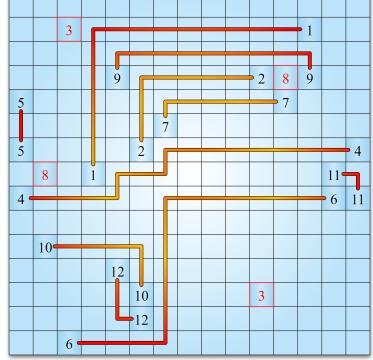
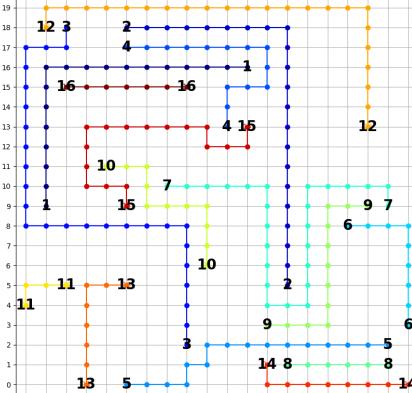
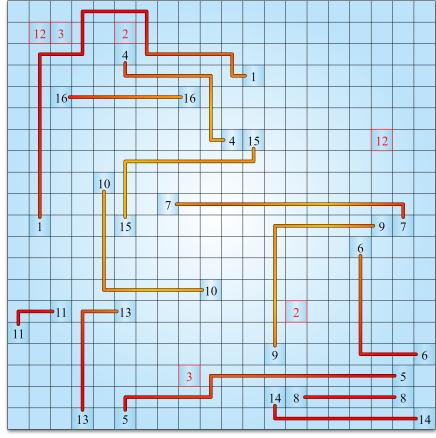
Nun möchte ich kurz die Zeitkomplexität des A\*-Algorithmus analysieren. Diese ist bei Weitem nicht trivial. Somit stellt sich zuerst die Frage, wie oft die `while`-Schleife maximal durchlaufen wird. Dazu müsste ermittelt werden, wie viele Zellen im ungünstigsten Fall in der Warteschlange landen könnten. Hierbei ist es wichtig zu beachten, dass **jede** freie und noch nicht besuchte Nachbarzelle des aktuellen Feldes in die Warteschlange eingefügt wird. In einem Gitter mit vielen Hindernissen könnte der Algorithmus gezwungen sein, zahlreiche Umwege zu nehmen. Dies würde dazu führen, dass mehr freie und unbesuchte Zellen in der Warteschlange landen. Die zentrale Fragestellung ist daher, wie viele solcher Zellen es in einem worst-case Szenario in einer Warteschlange geben könnte. Aufgrund der Komplexität der Fragestellung habe ich nachfolgend angenommen, dass die Zeitkomplexität der `while`-Schleife maximal  $O(b)$  beträgt, wobei  $b$  die maximale gesamte Anzahl der Zellen in der Warteschlange repräsentiert. Die Heuristik-Funktion hat eine konstante Zeitkomplexität von  $O(1)$ , da sie lediglich einfache arithmetische Operationen ausführt. Innerhalb der `while`-Schleife gibt es eine `for`-Schleife, die über die Nachbarn der aktuellen Zelle iteriert, und eine Überprüfung, ob ein Nachbar bereits in der `queue` ist. In einem Quadratgitter kann eine Zelle bis zu 4 Nachbarn haben, also ist die Zeitkomplexität dieser `for`-Schleife  $O(1)$  pro Zelle. Die Überprüfung, ob ein Nachbar in der `queue` ist, kann im schlimmsten Fall eine Zeitkomplexität von  $O(b)$  haben. Daher ergibt sich die Zeitkomplexität der `while`-Schleife zusätzlich der `for`-Schleife in der ungünstigsten Situation aus  $O(b^2)$ . Die Operationen `heappop` und `heappush` haben eine Zeitkomplexität von  $O(\log k)$ , wobei  $k$  die Anzahl der Elemente in der Warteschlange ist. Im schlimmsten Fall ist  $k = b$ , was zu einer Zeitkomplexität von  $O(\log b)$  führt. Daher ist die gesamte Zeitkomplexität des A\*-Algorithmus in der ungünstigsten Situation  $O(b^2 \log b)$ .

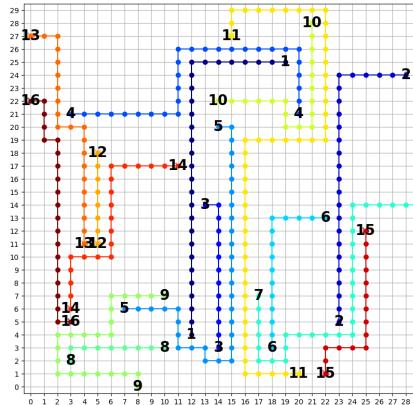
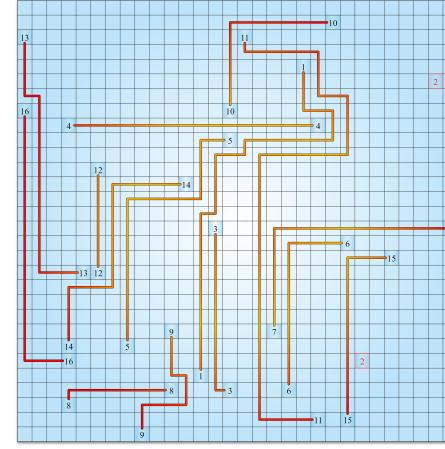
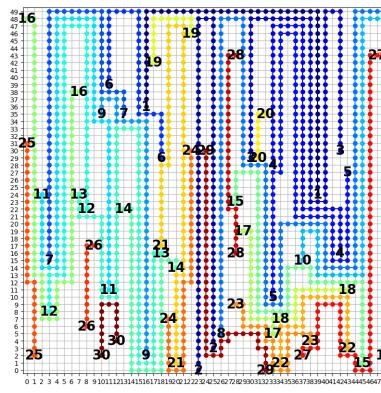
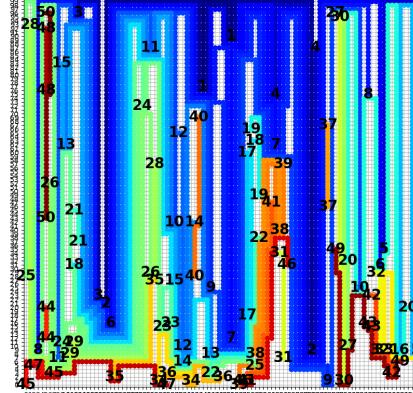
Darüber hinaus erstellte ich eine Methode `find_path`, welche den A\*-Algorithmus aufruft und sozusagen den Programmablauf koordiniert. Sie wählt zwei zufällige leere Zellen aus und überprüft, ob der gefundene Pfad mindestens eine Länge von drei Zellen besitzt. Abschließend habe ich eine Methode geschrieben, welche die gefundenen Pfade visualisiert. Innerhalb meiner `main`-Methode implementierte ich den Pseudocode, den ich schon in meiner Lösungsidee erläutert habe.

### 1.3 Beispiele

Nun möchte ich mich einigen Beispiel widmen. Die Eingabe besteht dabei aus der Ganzzahl  $N$ , wodurch anschließend die für das Erstellen benötigte Zeit und anschließend die Lösung des Rätsels visuell ausgegeben wird. Zuletzt wird ein Block bestehend aus  $N$ , der Anzahl an Linienzügen sowie dem eigentlichen Rätsel angezeigt. Somit dient dieser als Eingabe für den Arukone-Solver auf der Webseite des Bundeswettbewerbs Informatik. Nachfolgend habe ich einige Beispiele inklusive ihrer grafischen Ausgabe aufgelistet.

N	Lösung des erstellten Rätsels	Lösung des BWInf-Programms
4	 Zeit: 0.0 Sekunden	

N	Lösung des erstellten Rätsels	Lösung des BWInf-Programms
10	 <p>Zeit: 0.0039 Sekunden</p>	
15	 <p>Zeit: 0.013 Sekunden</p>	
20	 <p>Zeit: 0.017 Sekunden</p>	

N	Lösung des erstellten Rätsels	Lösung des BWInf-Programms
30	 Zeit: 0.54 Sekunden	
50	 Zeit: 0.66 Sekunden	Max. N=30
100	 Zeit: 12,43 Sekunden	Max. N=30

Bei den Beispielen fällt auf, dass das zur Verfügung gestellte Programm von BWInf nur sehr wenige Rätsel tatsächlich lösen kann. Schon ab  $N = 15$  löst der Arukone-Solver nur noch einen sehr geringen Bruchteil der generierten Rätsel.

## 1.4 Anhang

### 1.4.1 Quellcode

```

1 # Robert Vetter
# 20.10.2023
3 # Aufgabe 1 des 42. BWInf

5 # notwendigen Bibliotheken importieren
import random
7 import numpy as np
from collections import namedtuple
9 from heapq import heappop, heappush
import time
11 import matplotlib.pyplot as plt
import math

13 # Gitter initialisieren, 2D Array mit Nullen
15 def initialize_grid(N):
    return np.zeros((N, N), dtype=int)

17 # nimmt eine Zelle und N entgegen und gibt max. 4 moegliche Nachbarn zurueck, wenn diese
# innerhalb des Gitters liegen
19 def get_neighbours(cell, N):
    # Definiere die vier Richtungen: oben, unten, links und rechts
21     directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

23     # Koordinaten beachbarter Zellen
25     neighbours = []

27     # alle Richtungen durchgehen
29     for d in directions:
        # Koordinaten beachbarter Zelle berechnen und hinzufuegen
31         neighbour_x = cell[0] + d[0]
        neighbour_y = cell[1] + d[1]
        neighbours.append((neighbour_x, neighbour_y))

33     valid_neighbours = []
35     for i, j in neighbours:
        # innerhalb der Grenzen des Gitters
        if 0 <= i < N and 0 <= j < N:
            valid_neighbours.append((i, j))

39     # alle Nachbarn zurueckgeben
41     return valid_neighbours

43     # Anzahl an beachbarten Zellen, die noch leer sind
43 def get_empty_neighbours(grid, cell, N):
    # Liste der beachbarten Zellen
45     neighbours = get_neighbours(cell, N)

47     # Initialisiere den leeren Nachbarzaehler auf 0
49     empty_neighbour_count = 0

51     # Ueberpruefe jeden Nachbarn
52     for neighbour in neighbours:
        # wenn Wert des Nachbarn im Raster 0 ist, erhoehe den leeren Nachbarzaehler um 1
53         if grid[neighbour] == 0:
            empty_neighbour_count += 1

55     return empty_neighbour_count

57     # heuristische Funktion: Manhattan-Distanz zwischen zwei Punkten
59 def heuristic(a, b):
    # Unterschied zwischen den x-Koordinaten der Zellen a und b
61     x_difference = abs(a[0] - b[0])

63     # Unterschied zwischen den y-Koordinaten der Zellen a und b
64     y_difference = abs(a[1] - b[1])

65     return x_difference + y_difference

67     #Hauptalgorithmus fuer kuerzesten Weg zwischen zwei Zellen

```

```

69 def astar(grid, start, end, N, max_queue_size=1000):
70     # Tupelklasse "Node" mit den Feldern "total", "cell", "cost" und "heuristic"
71     Node = namedtuple("Node", ["total", "cell", "cost", "heuristic"])
72
73     # Set fuer besuchte Zellen
74     visited = set()
75
76     # Prioritaetswarteschlange (aufgrund von heappush und heappop) mit dem Startknoten und
77     # seiner heuristischen Entfernung zum Ziel
78     queue = [Node(heuristic(start, end), start, 0, heuristic(start, end))]
79
80     # Dictionary, um die Vorgaengerzellen zu speichern
81     came_from = {}
82
83     # solange die Warteschlange nicht leer ist
84     while queue:
85         # Entferne und erhalte Knoten mit niedrigsten Gesamtwert aus Warteschlange
86         current = heappop(queue)
87
88         # Wenn aktuelle Knoten Ziel, rekonstruiere Pfad und gebe ihn zurueck
89         if current.cell == end:
90             path = [end]
91             while end in came_from:
92                 end = came_from[end]
93                 path.append(end)
94             return path[::-1]
95
96         # Fuege aktuelle Zelle zum Set der besuchten Zellen
97         visited.add(current.cell)
98
99         # Gehe durch alle Nachbarn der aktuellen Zelle
100        for neighbour in get_neighbours(current.cell, N):
101            # Ueberspringe Nachbarn, wenn er bereits besucht wurde oder wenn er nicht leer ist
102            if neighbour in visited or grid[neighbour] != 0:
103                continue
104
105            # Berechne neuen Kosten fuer den Nachbarn
106            new_cost = current.cost + 1
107
108            # Erstelle neuen Knoten fuer Nachbarn mit den berechneten Kosten und Heuristik
109            new_node = Node(new_cost + heuristic(neighbour, end), neighbour, new_cost,
110                            heuristic(neighbour, end))
111
112            # Wenn der Nachbar noch nicht besucht wurde oder ein kuerzerer Pfad gefunden wurde
113            # fuege ihn zur Warteschlange hinzu und aktualisiere 'came_from'
114            if neighbour not in [item.cell for item in queue]:
115                came_from[neighbour] = current.cell
116                heappush(queue, new_node)
117
118        # Wenn kein Pfad gefunden, gebe eine leere Liste zurueck
119        return []
120
121    # astar-Methode auf zwie zufaellige Zellen aufrufen
122    def find_path(grid, current_path, N):
123        # Liste der leeren Zellen im Gitter erstellen
124        empty_cells = list(zip(*np.where(grid == 0)))
125
126        # Wenn keine leeren Zellen vorhanden sind, unveraendertes Gitter zurueckgeben
127        if not empty_cells or len(empty_cells) == 1:
128            return grid, []
129
130        # Ueberpruefen, ob alle verbleibenden leeren Zellen keine freien Nachbarn haben
131        if all(get_empty_neighbours(grid, cell, N) == 0 for cell in empty_cells):
132            return grid, []
133
134        # Start- und Endzellen zufaellig auswaehlen
135        start = random.choice(empty_cells)
136        end = random.choice(empty_cells)
137
138        # Neue Start- und Endzellen auswaehlen, wenn sie gleich sind oder keine leeren Nachbarn
139        # haben
140        while start == end or not (get_empty_neighbours(grid, start, N) and get_empty_neighbours(
141

```

```

        grid, end, N)):
    start = random.choice(empty_cells)
    end = random.choice(empty_cells)

141    # A* Algorithmus verwenden, um den Pfad zwischen Start und Ende zu finden
path = astar(grid, start, end, N)

143    # Wenn der Pfad gueltig ist und >= 3, Pfad zum Gitter hinzufuegen und aktuellen Pfad
zuweisen
145    if len(path) >= 3:
        for cell in path:
            grid[cell] = current_path
        return grid, path # Aktualisiertes Gitter und Pfad zurueckgeben

149    # Wenn kein gueltiger Pfad gefunden wurde, unveraendertes Gitter und leere Liste
zurueckgeben
151    return grid, []

153 # Gitter reformatieren (sodass es im Format eines zu loesenden Raetsels erscheint)
def reformat_grid(grid, paths):
155    # Durchlaufe jeden Pfad in den gefundenen Pfaden
    for path in paths:
        # Bestimme Kopf und Ende des Pfades
        head, tail = path[0], path[-1]

159        # Setze alle Zellen des Pfades auf 0 zurueck, ausser Kopf und Ende
161        for pos in path[1:-1]:
            grid[pos] = 0

163        # Setze Kopf und Ende gleich
165        grid[head] = grid[tail]

167    return grid

169 # grafische Ausgabe der Linienzuege
def visualize_paths(grid, paths):
171    N = grid.shape[0]
    plt.figure(figsize=(10, 10))

173    # Farbliste fuer die Paths
    colors = plt.cm.jet(np.linspace(0, 1, len(paths)))

175    # Sortiert Pfade basierend auf der Pfad-Nummer
    paths = sorted(paths, key=lambda path: grid[path[0]]))

177    # Zeichnet die Zellnummern und Pfad-Nummern
179    for i in range(N):
        for j in range(N):
            if grid[i, j] != 0:
                plt.text(j, N - 1 - i, str(grid[i, j]), ha='center', va='center', color=colors
[grid[i, j] - 1])

181    # Zeichnet Verbindungen zwischen den Zellen eines Pfades
183    for idx, path in enumerate(paths):
        for i in range(len(path)-1):
            start = path[i]
            end = path[i+1]
            plt.plot([start[1], end[1]], [N - 1 - start[0], N - 1 - end[0]], '-o', color=
colors[grid[path[0]] - 1])

185        # markiert Kopf und Ende mit fetter Schrift
187        head, tail = path[0], path[-1]
        link_number = grid[head]
        plt.text(head[1], N - 1 - head[0], str(link_number), ha='center', va='center', color='
black', weight='bold', fontsize=20)
189        plt.text(tail[1], N - 1 - tail[0], str(link_number), ha='center', va='center', color='
black', weight='bold', fontsize=20)

191    # plottet Pfade
193    plt.xlim(-0.5, N - 0.5)
    plt.ylim(-0.5, N - 0.5)
    plt.grid(True)
201    plt.xticks(np.arange(N))

203

```

```

205     plt.yticks(np.arange(N))
206     plt.show()

207
208     def main():
209         # Eingabe der Groesse des Gitters (N) vom Benutzer
210         print("Geben Sie die Laenge der Seite 'N' des Gitters an: ")
211         N = int(input())
212         # Ueberpruefung, ob N groesser oder gleich 4 ist, wenn nicht, wird Benutzer zur erneuten
213         # Eingabe aufgefordert
214         while N < 4:
215             print("Geben Sie ein groesseres N ein (N > 4): ")
216             N = int(input())

217         # Startzeit fuer die Zeitmessung
218         start = time.time()
219         # Maximale Anzahl von Pfaden, die erstellt werden koennen
220         max_links = 2 * N

221         # Initialisierung des Gitters
222         grid = initialize_grid(N)
223         # Initialisierung des aktuellen Flows
224         current_path = 1

225         # Liste zur Speicherung der Pfade
226         paths = []

227         if N % 2 == 0:
228             limit = N // 2
229         else:
230             limit = math.ceil(N / 2)

231         # Schleife zur Erstellung der Pfade, bis die Haelfte der Flows erreicht ist
232         while current_path < limit:
233             # Zuruecksetzen des aktuellen Pfaes und der gesamten Pfade fuer jede Iteration
234             current_path = 1
235             paths = []
236             # Zuruecksetzen des Gitters fuer jede Iteration
237             grid = initialize_grid(N)
238             # Schleife zur Erstellung der Pfade
239             for _ in range(max_links):
240                 grid, path = find_path(grid, current_path, N)
241                 # Wenn ein gueltiger Pfad gefunden wurde, wird er zur Liste der Pfade hinzugefuegt
242                 # und der aktuelle Pfad wird erhoeht
243                 if path:
244                     paths.append(path)
245                     current_path += 1

246         # Ausgabe der benoetigten Zeit zur Erstellung der Pfade
247         print("Benoetigte Zeit: ", (time.time() - start), " Sekunden")

248         # Ausgabe der moeglichen Loesung
249         print("Moegliche Loesung: \n")
250         for row in grid:
251             print(' '.join(map(str, row)))
252         # Kopie des Gitters zur spaeteren Verwendung
253         gridCopy = grid

254         # Ausgabe des Puzzles nach der Umformatierung des Gitters
255         print("Puzzle: \n")
256         grid = reformat_grid(grid, paths)
257         # Ausgabe der Groesse des Gitters und der Anzahl der Pfade
258         print(N)
259         print(current_path-1)
260         # Ausgabe des umformatierten Gitters
261         for row in grid:
262             print(' '.join(map(str, row)))
263         # Visualisierung der vollstaendigen Pfade
264         visualize_paths(gridCopy, paths)

265     if __name__ == "__main__":
266         main()

```