Aufgabe 1: Arukone

Niels Heiden
Team-ID: 00380
Team-Name: CTRL-C / CTRL-V

20. November 2023

Inhaltsverzeichnis

1 Lösungsidee **2**
1.1 Platzierung 2

2 Umsetzung **2**
2.1 Beispiele 5
2.2 Quellcode 6

1 Lösungsidee

Um lösbare Arukone-Rätsel zu erstellen, müssen Zahlenpaare nur an Stellen gesetzt werden, an denen sich auch eine Verbindung finden lässt. Das bedeutet, dass beim Setzen eines Zahlenpaares einfach nur überprüft werden muss, ob ein Weg ohne Kreuzung eines anderen Weges findbar ist.

1.1 Platzierung

Die einfachste Lösung besteht darin, jeder Zahl eines Zahlenpaares eine zufällige x,y-Koordinate zuzuordnen und dann mit einem Pfadfindungsalgorithmus einen Pfad zwischen diesen Koordinaten zu finden. Um dies effizient durchzuführen, ist die Verwendung von A* zur Wegfindung sinnvoll, da dieser Algorithmus die unnötigen Überprüfungen, die bei einem DFS-Algorithmus auftreten würden, erheblich reduziert. Im Worst-Case-Szenario bedeutet das eine Laufzeit von $O(n^2)$;dieser Worst-Case tritt auf, wenn die Zahlen in ungünstigen Positionen oder nah beieinander platziert sind. Dies liegt daran, dass bei Zahlen, die nah beieinander platziert sind, weniger Flexibilität für Pfade besteht und eine höhere Kreuzungsgefahr entsteht, was zu längeren Pfaden führt. Die durchschnittliche Laufzeit liegt in etwa bei $O(n \log n)$ und im besten Fall, wenn wenige Hindernisse im Weg sind, nähert sich die Laufzeit an lineare Komplexität an.

2 Umsetzung

```
excluded = []
paths = []
```

Globale Listen namens excluded und paths werden eingerichtet, um Positionen, die nicht mehr verfügbar sind, und bereits gefundene Pfade zwischen den Zahlenpaaren zu speichern.

```
def is_valid_position(puzzle, x, y):
    #check if the cell is within the grid and is empty and is not part of an existing path
    return puzzle[x][y] == 0 and (x, y) not in excluded
```

Die Funktion is_valid_position überprüft, ob eine gegebene Position auf dem Gitter zugelassen ist. Das bedeutet, sie darf noch nicht von einem Pfad belegt sein und muss innerhalb des Gitters liegen.

```
def generator(n, numCount): #generate a puzzle
    puzzle = [[0 for _ in range(n)] for _ in range(n)] #create an empty grid
    for num in range(1, numCount + 1): #for each number
        placed = False #boolean to indicate if the number has been placed
        for _ in range(n*n):  #attempts scaled to the size of the puzzle
            x1, y1 = random.randint(0, n-1), random.randint(0, n-1) #get random coordinates for the first number
            x2, y2 = random.randint(0, n-1), random.randint(0, n-1) #get random coordinates for the second number
            if (x1, y1) != (x2, y2) and is_valid_position(puzzle, x1, y1) and is_valid_position(puzzle, x2, y2) and doesPathExist(puzzle, (x1, y1), (x2, y2), excluded): #if the coordinates are not the same and are valid and a path exists between them (using A*)
                puzzle[x1][y1] = num #place the number at the first coordinates
```

```
                puzzle[x2][y2] = num #place the number at the second coordinates
                placed = True #indicate that the number has been placed
                break #break out of the loop
        if not placed: #if the number was not placed
            return None  #return None to indicate that the number pair could not be placed (this will cause the puzzle to be regenerated)
    return puzzle, paths #return the puzzle and the paths
```

Die generator-Funktion ist der Hauptteil des Codes. Sie versucht, ein lösbares Puzzle zu generieren, indem sie für jedes Zahlenpaar mit dem Random modul zufällige Positionen auf dem Gitter wählt. Mithilfe des A*-Algorithmus wird dann überprüft, ob zwischen diesen Positionen ein gültiger Pfad existiert, der keine anderen Pfade kreuzt. Wenn die positionen gültig sind und ein Pfad existiert werden die nummern an den Koordinaten platziert; Wenn nicht werden neue Koordinaten geprüft, dies kann maximal n^2 mal passieren, da sonst die selbe Position mehrmals probiert werden kann. Meine Funktion wäre noch durch die einführung einer already_tried Optimierbar. Wenn keine Position gefunden wurde wird die generierung abgebrochen.

```
def saveAsTxt(puzzle, n, numCount):
    f = open("arukone/exports/puzzle.txt", "w")
    f.write(str(n) + "\n")
    f.write(str(numCount) + "\n")
    for row in puzzle:
        f.write(' '.join(map(str, row)) + "\n")
    f.close()

#save puzzle and solution to .json file for drawing
def saveAsJson(puzzle, n, numCount):
    data = {
        "puzzle": puzzle,
        "solution": paths
    }

    with open("arukone/exports/puzzle.json", "w") as f:
        json.dump(data, f)
```

Sobald das Rätsel generiert ist, bieten die Funktionen saveAsTxt und saveAsJson Möglichkeiten, das Puzzle in verschiedenen Formaten zu speichern. Diese Funktionalität ist praktisch, um die generierten Rätsel zu speichern und für spätere Zwecke verfügbar zu haben.

```
#A* Algorithm Implementation to check if a path exists between two points
def heuristic(a, b):
    #Manhattan distance on a square grid
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def doesPathExist(puzzle, start, end, excluded):
    n = len(puzzle) #size of the puzzle
    open_set = set([start]) #set of nodes to be evaluated
    came_from = {} #map of navigated nodes
    g_score = {start: 0} #cost from start along best known path
    f_score = {start: heuristic(start, end)} #estimated total cost from start to goal through y

    while open_set: #while there are still nodes to be evaluated
        current = min(open_set, key=lambda x: f_score[x]) #node in open_set having the lowest f_score[] value

        if current == end: #if the current node is the end node
```

```
            path = reconstruct_path(came_from, current) #reconstruct the path
            excluded.extend(path) #add the path to the excluded list
            paths.append(path) #add the path to the paths list
            return True #return true to indicate that a path exists

        open_set.remove(current) #remove the current node from the open_set
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]: #for each neighbor of the current node
            neighbor = (current[0] + dx, current[1] + dy) #get the neighbor's coordinates
            if 0 <= neighbor[0] < n and 0 <= neighbor[1] < n and neighbor not in excluded: #if the neighbor is within the grid and is not part of an existing path
                tentative_g_score = g_score[current] + 1 #calculate the tentative g_score
                if tentative_g_score < g_score.get(neighbor, float("inf")): #if the tentative g_score is less than the neighbor's g_score
                    came_from[neighbor] = current #set the neighbor's came_from to the current node
                    g_score[neighbor] = tentative_g_score #set the neighbor's g_score to the tentative g_score
                    f_score[neighbor] = tentative_g_score + heuristic(neighbor, end) #set the neighbor's f_score to the tentative g_score + the heuristic
                    if neighbor not in open_set: #if the neighbor is not in the open_set
                        open_set.add(neighbor) #add the neighbor to the open_set

    return False #return false because no path was found

def reconstruct_path(came_from, current): #reconstruct the path
    total_path = [current] #start with the current node
    while current in came_from: #while the current node is in the came_from map
        current = came_from[current] #set the current node to the node that came before it
        total_path.append(current) #add the current node to the total path
    return total_path[::-1] #return the total path in reverse order (start to end)
```

Die Funktionen heuristic, doesPathExist und reconstructPath implementieren den A*-Algorithmus, der die Pfadfindung zwischen Zahlenpaaren ermöglicht. Der Algorithmus wählt den Pfad, der das vielversprechendste Verhältnis von Kosten (bereits aufgewendet) und potenziellem Aufwand (zum Ziel) bietet. Er fährt fort, die vielversprechendsten Pfade zu erkunden, bis das Ziel gefunden ist. Wege, die zu höheren Gesamtkosten führen würden, werden vermieden. Dadurch kann A* effizient den kürzesten Weg finden, ohne alle möglichen Wege untersuchen zu müssen.

```
def main(n):
    numCount = random.randint(n//2, n)
    puzzle, path = generator(n, numCount)
    if puzzle is None:
        print('Failed to generate a puzzle')
    else:
        print_puzzle(puzzle, n, numCount)
        saveAsTxt(puzzle, n, numCount)
        saveAsJson(puzzle, n, numCount)
```

```python

if __name__ == '__main__':
    time_start = time.perf_counter()
    main(30)
    time_end = time.perf_counter()
    print('{:5.3f}s'.format(time_end-time_start), end='  ')
    draw_puzzle()
```

Die Hauptfunktion main steuert den Erstellungsprozess, indem sie die Größe des Puzzles bestimmt und den Generator aufruft. Wenn ein Puzzle erfolgreich generiert wird, wird es gespeichert und ausgegeben.

2.1 Beispiele

Beispiele für Arukone Rätsel mit $n = 10, 20$ oder 30. Die Lösung meines Programms befindet sich immer links und die Lösung oder der Lösungsversuch des BWINF Programmns immer rechts.

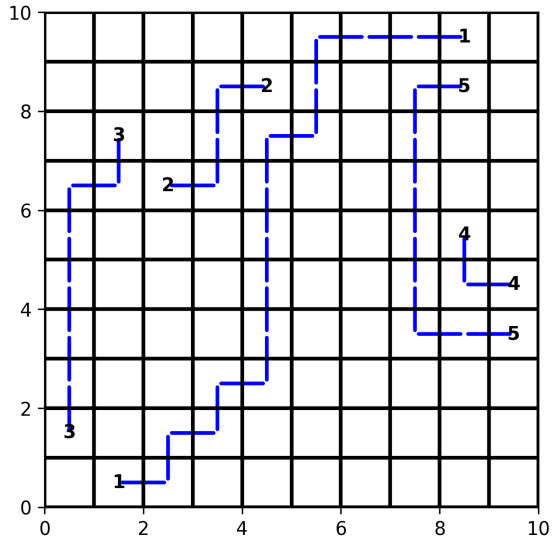

Abbildung 1: Optimale Lösung

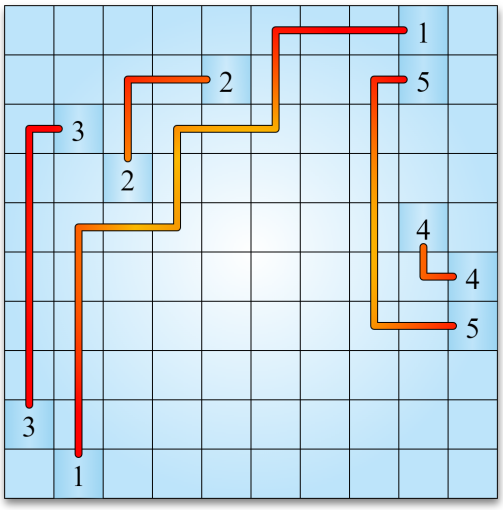

Abbildung 2: Lösung vom BWINF Programm

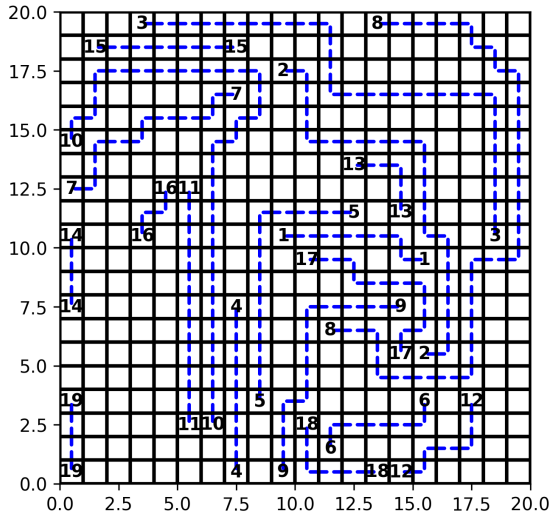

Abbildung 3: Beweis das Lösung möglich

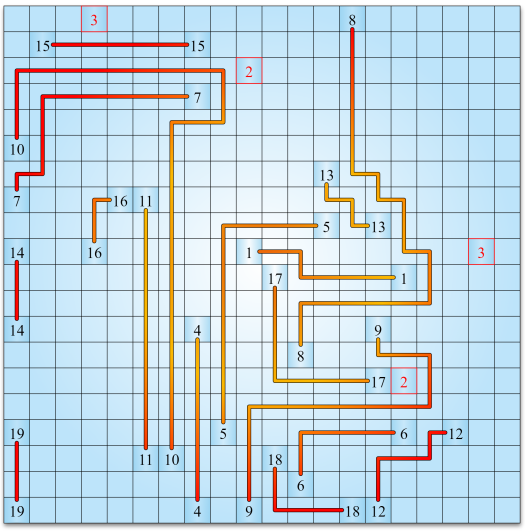

Abbildung 4: Lösungsversuch

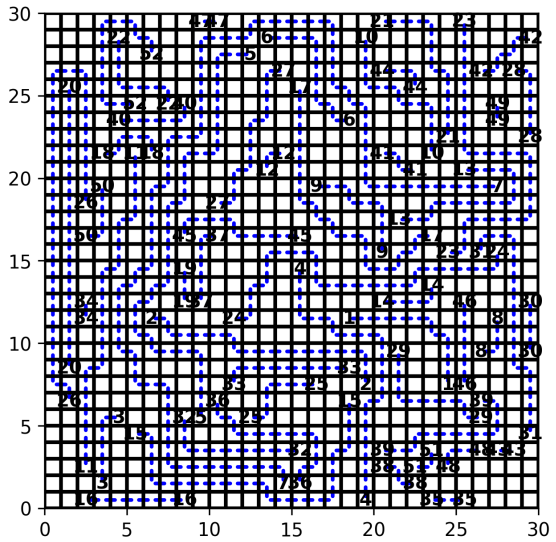

Abbildung 5: Beweis das Lösung möglich

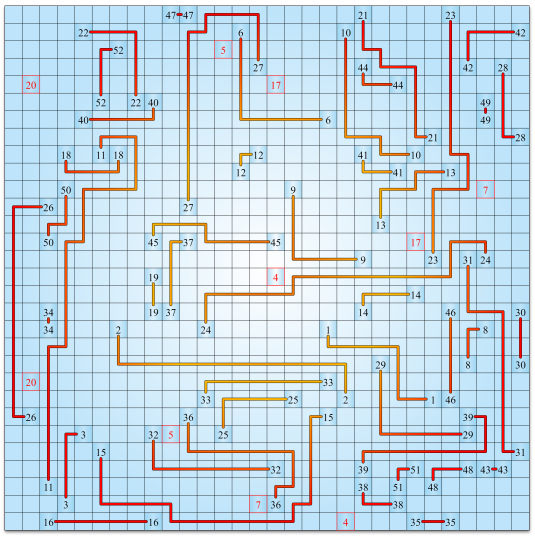

Abbildung 6: Lösungsversuch

2.2 Quellcode

```
import random #for generating random numbers
from collections import deque
import math
import json #for saving the puzzle and solution to a .json file
from drawArukone import main as draw_puzzle

excluded = []
paths = []

def is_valid_position(puzzle, x, y):
    #check if the cell is within the grid and is empty and is not part of an existing path
    return puzzle[x][y] == 0 and (x, y) not in excluded

def generator(n, numCount): #generate a puzzle
    puzzle = [[0 for _ in range(n)] for _ in range(n)] #create an empty grid
    for num in range(1, numCount + 1): #for each number
        placed = False #boolean to indicate if the number has been placed
        for _ in range(n*n):  #attempts scaled to the size of the puzzle
            x1, y1 = random.randint(0, n-1), random.randint(0, n-1) #get random coordinates for the first number
            x2, y2 = random.randint(0, n-1), random.randint(0, n-1) #get random coordinates for the second number
            if (x1, y1) != (x2, y2) and is_valid_position(puzzle, x1, y1) and is_valid_position(puzzle, x2, y2) and doesPathExist(puzzle, (x1, y1), (x2, y2), excluded): #if the coordinates are not the same and are valid and a path exists between them (using A*)
                puzzle[x1][y1] = num #place the number at the first coordinates
                puzzle[x2][y2] = num #place the number at the second coordinates
                placed = True #indicate that the number has been placed
                break #break out of the loop
```

```python
        if not placed: #if the number was not placed
            return None  #return None to indicate that the number pair could not be placed (this will cause the puzzle to be regenerated)
    return puzzle, paths #return the puzzle and the paths


#save functions
#save puzzle to file txt
def saveAsTxt(puzzle, n, numCount):
    f = open("arukone/exports/puzzle.txt", "w")
    f.write(str(n) + "\n")
    f.write(str(numCount) + "\n")
    for row in puzzle:
        f.write(' '.join(map(str, row)) + "\n")
    f.close()

#save puzzle and solution to .json file for drawing
def saveAsJson(puzzle, n, numCount):
    data = {
        "puzzle": puzzle,
        "solution": paths
    }

    with open("arukone/exports/puzzle.json", "w") as f:
        json.dump(data, f)

#print functions
def print_puzzle(puzzle, n, numCount):
    print(n)
    print(numCount)
    for row in puzzle:
        print(' '.join(map(str, row)))

def print_excluded(excluded):
    print(excluded)

#A* Algorithm Implementation to check if a path exists between two points
def heuristic(a, b):
    #Manhattan distance on a square grid
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def doesPathExist(puzzle, start, end, excluded):
    n = len(puzzle) #size of the puzzle
    open_set = set([start]) #set of nodes to be evaluated
    came_from = {} #map of navigated nodes
    g_score = {start: 0} #cost from start along best known path
    f_score = {start: heuristic(start, end)} #estimated total cost from start to goal through y

    while open_set: #while there are still nodes to be evaluated
        current = min(open_set, key=lambda x: f_score[x]) #node in open_set having the lowest f_score[] value

        if current == end: #if the current node is the end node
            path = reconstruct_path(came_from, current) #reconstruct the path
            excluded.extend(path) #add the path to the excluded list
            paths.append(path) #add the path to the paths list
```

```python
            return True #return true to indicate that a path exists

        open_set.remove(current) #remove the current node from the open_set
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]: #for each neighbor of the current node
            neighbor = (current[0] + dx, current[1] + dy) #get the neighbor's coordinates
            if 0 <= neighbor[0] < n and 0 <= neighbor[1] < n and neighbor not in excluded: #if the neighbor is within the grid and is not part of an existing path
                tentative_g_score = g_score[current] + 1 #calculate the tentative g_score
                if tentative_g_score < g_score.get(neighbor, float("inf")): #if the tentative g_score is less than the neighbor's g_score
                    came_from[neighbor] = current #set the neighbor's came_from to the current node
                    g_score[neighbor] = tentative_g_score #set the neighbor's g_score to the tentative g_score
                    f_score[neighbor] = tentative_g_score + heuristic(neighbor, end) #set the neighbor's f_score to the tentative g_score + the heuristic
                    if neighbor not in open_set: #if the neighbor is not in the open_set
                        open_set.add(neighbor) #add the neighbor to the open_set

    return False #return false because no path was found

def reconstruct_path(came_from, current): #reconstruct the path
    total_path = [current] #start with the current node
    while current in came_from: #while the current node is in the came_from map
        current = came_from[current] #set the current node to the node that came before it
        total_path.append(current) #add the current node to the total path
    return total_path[::-1] #return the total path in reverse order (start to end)

def main(n):
    numCount = random.randint(n//2, n)
    puzzle, path = generator(n, numCount)
    if puzzle is None:
        print('Failed to generate a puzzle')
    else:
        print_puzzle(puzzle, n, numCount)
        saveAsTxt(puzzle, n, numCount)
        saveAsJson(puzzle, n, numCount)

if __name__ == '__main__':
    main(20)
    draw_puzzle()
```