Dokumentation des Labyrinth-Projekts in MESA

Paul Zintl, Luca Wolter

June 28, 2024

Contents

1 Einführung

Dieses Projekt beschreibt die Erstellung und Simulation eines Labyrinths mithilfe der MESA-Bibliothek in Python. Ziel ist es, einen Agenten durch das Labyrinth zu navigieren und anschließend soziale Interaktionen zwischen Agenten zu simulieren.

2 Projektbeschreibung

1. Erstellen Sie ein Labyrinth in MESA.
2. Platzieren Sie ein Ziel und einen sich bewegenden Agenten in dem Labyrinth.
3. Erstellen Sie aus der Position des Agenten und des Ziels sowie der Topologie des Labyrinths einen Graphen für das Suchproblem. Visualisieren Sie den Graphen und speichern Sie ihn als Bild.
4. Lösen Sie die Suchaufgabe mit A*.
5. Drucken Sie den Pfad P, seine Länge L und die Kosten C aus und aktualisieren Sie das Graphenbild.
6. Bewegen Sie Ihren Agenten entsprechend dem Pfad zum Ziel (ein Schritt ist die Bewegung des Agenten).
7. Wenn der Agent das Ziel erreicht, ist er reich! Geben Sie ihm 10 Münzen.
8. Entfernen Sie die Elemente des Labyrinths und erstellen Sie L-1 reiche Agenten mit 10 Münzen und L arme Agenten mit 0 Münzen. Lassen Sie die Agenten sich zufällig im Gitter bewegen.
9. Wenn der reiche Agent mindestens 3 arme Agenten mit weniger als 3 Münzen als Nachbarn hat und selbst mehr als 5 Münzen besitzt, teilt er sein Geld mit einem zufällig ausgewählten armen Nachbarn. Wenn der arme Agent nach 5 Schritten kein Geld hat, muss er sterben (bleibt stehen, wechselt seine Farbe und interagiert mit den anderen nicht mehr).
10. Die Simulation hält nach 20 Schritten an. Visualisieren Sie die Statistik (wie viele Agenten 0..10 Münzen haben, wie viele Agenten lebendig/tot sind) für jeden Simulationsschritt.

3 Implementierung

3.1 Klassendiagramm

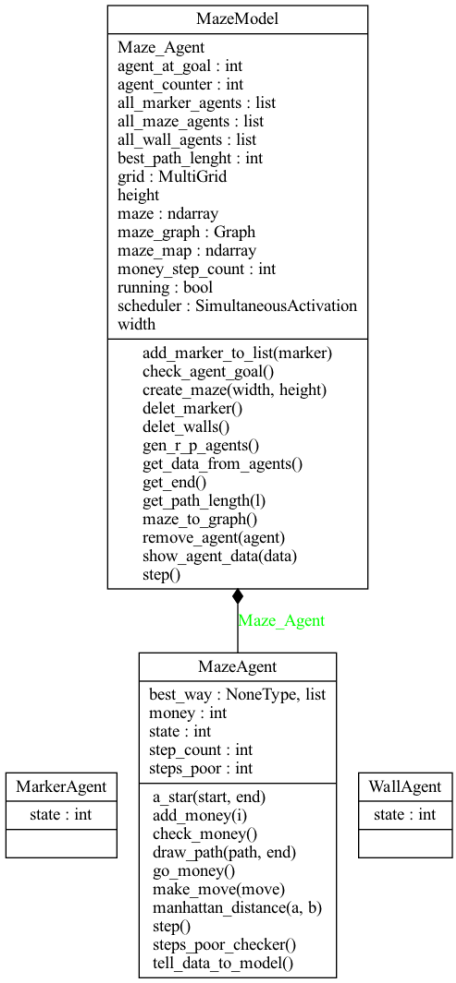


Figure 1: Caption
Unser Klassenmodel, welches wir für das Projekt verwenden.

3.2 Erstellung des Labyrinths

Listing 1: Erstellung des Labyrinths

```
import random
import numpy as np

def create_maze(width, height):
    maze = [[1 for _ in range(width)] for _ in range(height)]
    directions = [(0, 2), (0, -2), (2, 0), (-2, 0)]

    def is_valid(x, y):
        return 0 < x < height-1 and 0 < y < width-1

    def carve_path(x, y):
        maze[x][y] = 0
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny) and maze[nx][ny] == 1:
                maze[nx-dx//2][ny-dy//2] = 0
                carve_path(nx, ny)
    carve_path(1, 1)
    maze[1][1] = 0
    return maze

print_maze(create_maze(15, 15))
```

Der Code erstellt ein Labyrinth mithilfe von rekursiver Pfadgenerierung. Diese Funktion erstellt ein Labyrinth mit einer bestimmten Breite (width) und Höhe (height).Ein zweidimensionales Array (maze = [[1 for - in range(width)] for - in range(height)]) wird erstellt, wobei jede Zelle des Arrays den Wert 1 hat. Der Wert 1 repräsentiert eine Wand. Diese Hilfsfunktion (is-valid) überprüft, ob die gegebenen Koordinaten (x, y) innerhalb der gültigen Grenzen des Labyrinths liegen. Dadurch wird sichergestellt, dass keine Bewegung außerhalb der Grenzen des Arrays erfolgt. Die Funktion carve-maze generiert einen Pfad im Labyrinth. Die Bewegungsrichtungen werden zufällig gewählt um den Pfad zu ebnen. Für jede Richtung wird eine neue Position berechnet. Wenn die neue Position noch ein Wand-Agent ist, wird die Wand zwischen der aktuellen und der neuen Position entfernt. Der Pfad beginnt bei (1,1) um sicherzustellen, dass das Labyrinth einen Startpunkt hat.

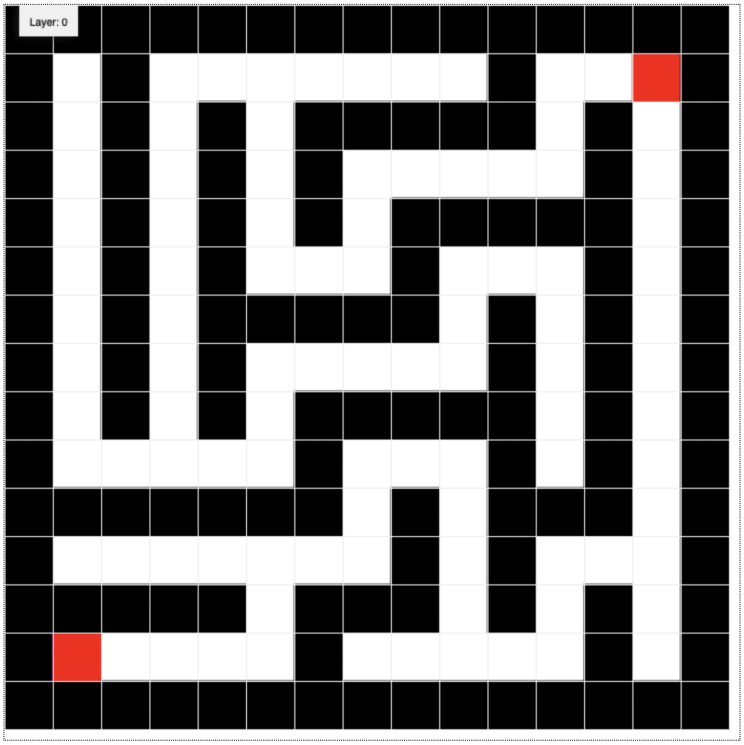


Figure 2: Caption
Beispielergebnis. Der rote Kasten unten links ist der Start-Punkt und der rote Kasten oben rechts ist das Ende des Labyrinthes. Beides ist immer Festgegeben.

3.3 Platzierung vom Labyrinth

Listing 2: Platzierung von Agent und Ziel

```
from maze_model import MarkerAgent, MazeAgent, mesa, MazeModel

grid = mesa.visualization.CanvasGrid(agent_portrayal, 15, 15, 1000, 1000)
server = mesa.visualization.ModularServer(
    MazeModel,
    [grid],
    "Maze-Model",
    {"width": 15, "height": 15}
)
server.port = 8521
server.launch()
```

In diesem Abschnitt wird das Labyrinth erstellt und über eine Server mit dem Standart-Port geladen. Aus die Groeße 700 x 700 Pixeln und 15 Agenten breit und hoch.

3.4 Erstellung des Graphen und Visualisierung

Listing 3: Erstellung des Graphen und Visualisierung

```
import networkx as nx
import matplotlib.pyplot as plt

def maze_to_graph(maze):
    graph = nx.Graph()
    width, height = maze.shape
    for x in range(width):
        for y in range(height):
            if maze[x, y] == 0:
                pos = (x, y)
                graph.add_node(pos)
                for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    neighbor = (x + dx, y + dy)
                    if 0 <= neighbor[0] < width and 0 <= neighbor[1] < height
                    and maze[neighbor[0], neighbor[1]] == 0:
                        graph.add_edge(pos, neighbor)
    return graph

maze = np.array(create_maze(15, 15))
graph = maze_to_graph(maze)

pos = {(x, y): (x, -y) for x, y in graph.nodes()}
plt.figure(figsize=(8, 8))
nx.draw(graph, pos, with_labels=True, node_size=700, node_color="lightblue",
font_size=10, font_weight="bold", edge_color='gray')
plt.gca().invert_yaxis()
plt.title("Maze Graph")
plt.show()
```

In diesem Abschnitt vom Code wird ein Graph erstellt, welcher alle Wege im Labyrinth beinhaltet. Die Länge des Graphen ist über die width und height des Labyrinthes vorbestimmt. Dabei wird das zweidimensionale Array durchgegangen und die Nachbarn verglichen. Wenn diese eine 0 sind (bei uns kein Wand Agent) wird ein neuer Knotenpunkt hinzugefügt. Danach werden die Knoten noch durch Kanten verbunden und der Graph wird zurückgegeben. Über nx.draw soll der Graph gezeichnet werden und mithilfe von plt.gca().invert-yaxis() wird die Y-Achse gespiegelt, da der Graph zuvor in die falsche Richtung verlaufen ist.

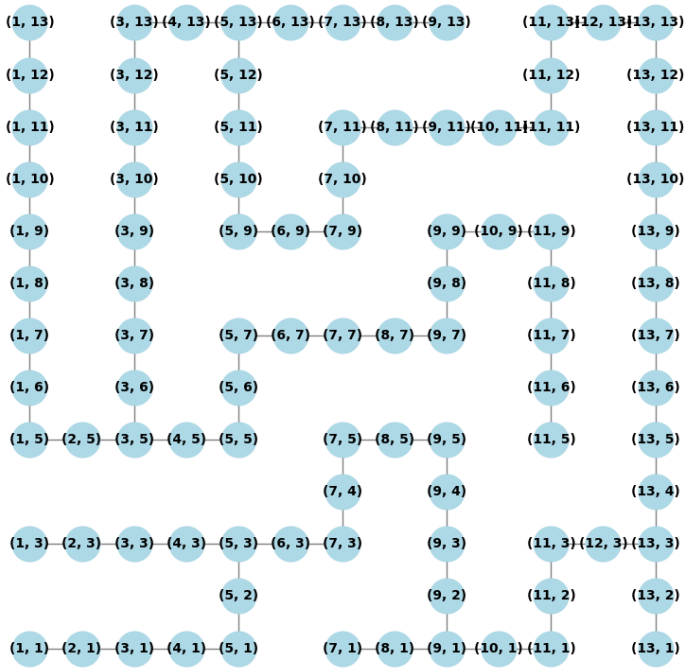


Figure 3: Caption
(1,1) ist der Anfang und (13,13) ist das Ende.

3.5 Lösung mit A*

Listing 4: Lösung mit A*

```
import math

def a_star(maze, start, end):
    G = maze_to_graph(maze)
    open_list = set([start])
    closed_list = set([])
    g = {start: 0}
    parents = {start: start}

    def manhattan_distance(a, b):
        return math.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)

    while len(open_list) > 0:
        n = None
```

```
            for v in open_list:
                if n == None or g[v] + manhattan_distance(v, end) < g[n] +
                manhattan_distance(n, end):
                    n = v

            if n == end or G[n] == end:
                path = []
                while parents[n] != n:
                    path.append(n)
                    n = parents[n]
                path.append(start)
                path.reverse()
                return path

            open_list.remove(n)
            closed_list.add(n)
            for (nx, ny) in G.neighbors(n):
                if (nx, ny) in closed_list:
                    continue
                candidate_g = g[n] + 1
                if (nx, ny) not in open_list or candidate_g < g[(nx, ny)]:
                    g[(nx, ny)] = candidate_g
                    parents[(nx, ny)] = n
                    if (nx, ny) not in open_list:
                        open_list.add((nx, ny))
    return None

start = (1, 1)
end = (13, 13)
path = a_star(maze, start, end)
print("Pfad:", path)
```

In diesem Abschnitt wird der A*-Algorithmus definiert. Dieser bekommt den Graphen, welcher das Array ist in, welchem sich der bestmögliche Weg zum Ende befindet. Die manhattan-distance Funktion ist unsere Heuristik. Diese berechnet die Luftlinie zum Zielpunkt von dem aktuellen Knotenpunkt der Liste. Die Länge des Graphen vom aktuellen Punkt wird in g gespeichert, durch zählen aller Knotenpunkte. Durch die manhattan-Funktion und g werden die Kosten addiert. Es wird außerdem eine Liste erstellt welche alle offenen Knotenpunkte beinhaltet, eine Liste erstellt welche die Nachbarknotenpunkte als Eltern ansieht und merkt welche als nächstes angeschaut werden müssen und eine geschlossene Liste erstellt, in welche alle Knoten kommen, die schon angeschaut wurden. Alle Kosten werden so lange berechnet bis es keine offenen Knoten mehr gibt. Am Ende geben wir noch das Array aus über die print-Funktion.

3.6 Graph des Besten Weges

Listing 5: Graph des Besten Weges

```
def draw_path(self, path, end):
        path_graph = nx.Graph()

    pos = {(x, y): (x, -y) for (x, y) in path_graph.nodes()}

        plt.figure(figsize=(8, 8))
        nx.draw(path_graph, pos, with_labels=False, node_size=700,
        node_color="lightblue", font_size=10, font_weight="bold",
        edge_color='gray')

        node_labels = nx.get_node_attributes(path_graph, 'cost')
        distance_labels = nx.get_node_attributes(path_graph, 'distance')
        formatted_labels = {
        node: f"C={cost}\nL={distance}"
        for node, cost, distance in zip(node_labels.keys(),
        node_labels.values(), distance_labels.values())
        }
        nx.draw_networkx_labels(path_graph, pos,
        labels=formatted_labels, font_size=10)

        plt.gca().invert_yaxis()
        plt.title("Path")
```

In dem Abschnitt wird der Graph ausgegeben, wecher oben berechnet wurde. Es werden nur die Knoten angegeben, welche sich im günstigsten Weg befinden. Der Graph wird nach jedem Schritt erneuert und somit werden die Knoten entfernt welche der Agent schon abgelaufen ist.

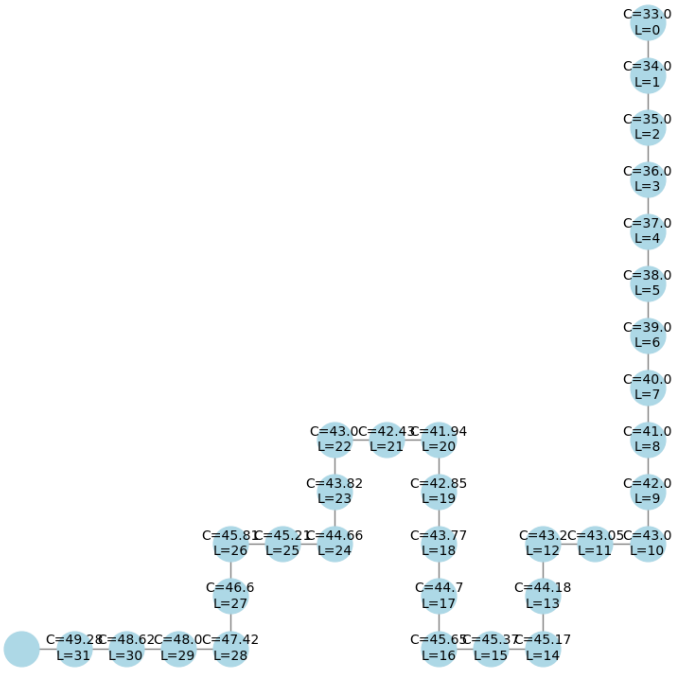


Figure 4: Caption
(1,1) ist der Anfang und (13,13) ist das Ende.

3.7 Agentenbewegung und Belohnung

Listing 6: Agentenbewegung und Belohnung

```
class MazeAgent(mesa.Agent):
    def __init__(self, model, id, state=0, money=0):
        super().__init__(id, model)
        self.state = state
        self.money = money
        self.best_way = []
        self.step_count = 0
        self.steps_poor = 0

    def step(self):
        if self.state == 2:
            pass
        elif self.state == 0:
            if not self.best_way:
```

```
                self.best_way = a_star(self.model.maze,
                self.pos, self.model.get_end())
                print(self.best_way)
            if self.step_count < len(self.best_way):
                self.make_move(self.best_way[self.step_count])
                self.step_count += 1
            else:
                self.model.get_path_length(len(self.best_way))
                self.state = 1
                self.money += 10
        elif self.state == 1:
            self.go_money()
            self.steps_poor_checker()

    def make_move(self, move):
        marker_agent = MarkerAgent(self.model, self.model.next_id(), 1)
        self.model.grid.place_agent(marker_agent, self.pos)
        self.model.add_marker_to_list(marker_agent)
        self.model.grid.move_agent(self, move)
```

In diesem Codeabschnitt ist jeder Schritt, den ein Agent macht definiert. In der Funktion –init– wird über die Vererbung model die gesammte Umgebung und alle anderen Agenten vererbt. Zudem wird die ID vererbt, welche für eindeutige Kennung der Agenten verwendet wird. Der state=0 wird übergeben solange der Agent noch nicht im Ziel ist. Das Atribut money ist für die Anfangsmenge an Geld, die der Agent besitzt (standardmäßig auf 0 gesetzt). In –init– wird auch der beste Weg gespeichert. In self-state wird der Zustand des Agenten angegeben, für die Aktion die der Agent ausführt. Self.best-way ist eine Liste, die den besten Weg speichert, den der A*-Algorithmus gefunden hat. Der super()-Aufruf ruft den Konstruktor der Basisklasse mesa.Agent auf, um sicherzustellen, dass alle notwendigen Initialisierungen der Basisklasse durchgeführt werden. Der Zweck des –init–Konstruktors in der MazeAgent-Klasse ist es, den Agenten mit den notwendigen Startwerten zu initialisieren und ihn in den Zustand zu versetzen, in dem er seine Simulationstätigkeiten aufnehmen kann. Das bedeutet, wenn du einen neuen MazeAgent erstellst, initialisiert der –init–Konstruktor diesen Agenten mit den angegebenen Werten für model, id, state und money und bereitet ihn auf seine Aufgaben vor. Die step-Methode definiert das Verhalten des Agenten basierend auf seinem Zustand: Nichts tun, wenn tot (Zustand 2), den besten Weg finden und diesem folgen (Zustand 0), oder sich zufällig bewegen und Geld abgeben (Zustand 1). Die step-Methode definiert das Verhalten des Agenten basierend auf seinem Zustand: Nichts tun, wenn tot (Zustand 2), den besten Weg finden und diesem folgen (Zustand 0), oder sich zufällig bewegen und Geld abgeben (Zustand 1).

3.8 Interaktionen zwischen den Agenten

Listing 7: Interaktionen zwischen den Agenten

```
def go_money(self):
    neighbors = self.model.grid.get_neighbors(self.pos, moore=False,
    include_center=False)
    if self.money >= 3:
        for neighbor in neighbors:
            if neighbor.state == 1 and neighbor.money < 3:
                self.give_money(neighbor)

def give_money(self, agent):
    self.money -= 1
    agent.money += 1

def steps_poor_checker(self):
    if self.money == 0:
        self.steps_poor += 1
        if self.steps_poor >= 5:
            self.state = 2
```

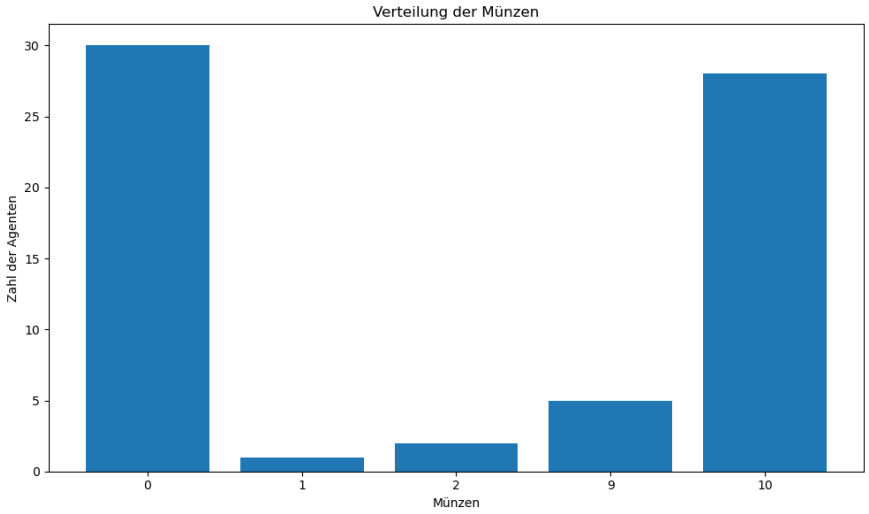
Nach dem Labyrinth abgelaufen wurde, werden alle Agenten entfernt bis auf den Agent, welcher das Ziel erreicht hat. Dieser erhällt zehn Münzen. Es werden nun neun reiche Münzen zufällig erzeugt und zehn arme Agenten, welche die reichen Agenten suchen um Geld zu bekommen. Der Code definiert das Verhalten eines Agenten in einer Multi-Agenten-Simulation in Bezug auf die Geldverteilung und die Überwachung von Geldmangel. Die Methode go-money ermöglicht es dem Agenten, Geld an bedürftige Nachbarn zu verteilen, wenn er genug Geld hat. Die Methode give-money führt die tatsächliche Geldübertragung durch. Die Methode steps-poor-checker überwacht, wie lange der Agent ohne Geld bleibt, und ändert seinen Zustand, wenn er für eine bestimmte Anzahl von Schritten, hier fünf Schritte, kein Geld hat. Diese Mechanismen steuern das Verhalten des Agenten und seine Reaktion auf anhaltenden Geldmangel.

3.9 Statistiken und Visualisierung

Listing 8: Statistiken und Visualisierung

```
def step(self):
    ...

    if self.money_step_count <= 20:
        self.scheduler.step()
    elif self.money_step_count == 21:
       self.show_agent_data(self.get_data_from_agents())
       self.running = False
    else:
        print(str(self.money_step_count) + " Steps sind vorbei")

def get_data_from_agents(self):
    agents_data = []
    for agent in self.all_maze_agents:
        agents_data.append(agent.tell_data_to_model())
        print(agent.tell_data_to_model())
    print(agents_data)
    return agents_data

def show_agent_data(self, data):
    counter = {}
    for item in data:
        tuple_item = tuple(item)
        if tuple_item in counter:
            counter[tuple_item] += 1
        else:
            counter[tuple_item] = 1

    combined = list(counter.items())

    sorted_combined = sorted(combined, key=lambda x: x[0])

    x_vals = [f'{x}' for x, _ in sorted_labels]
    y_vals = list(sorted_values)

    plt.figure(figsize=(10, 6))
    plt.bar(x_vals, y_vals)
    plt.xlabel('Muenzen')
    plt.ylabel('Zahl der Agenten')
    plt.title('Verteilung der Muenzen')
    plt.tight_layout()
    plt.show()
```

Diese Methode steuert den Fortschritt der Simulation. Sie führt die Schritte der Agenten aus, sammelt Daten und zeigt sie nach 20 Schritten an. In der step-Funktion werden die 20 Schritte gemacht. Beim 21. Schritt werden die Agenten-informationen gesammelt und ausgegeben. Die get-data-from-agents-Funktion sammelt die Daten von allen Agenten. Diese speichert für jeden Agenten das Ergebnis in deragents-data Liste. Counter, ein leeres Wörterbuch wird erstellt, um die Häufigkeit jeder Datenkombination zu zählen. Combined erstellt eine Liste der Items aus dem Wörterbuch counter, welche von sorted-combined nach den Münzen sotiert wird. 'x-vals' und 'y-vals' extrahieren die Münzen und deren Häufigkeiten für das Diagramm. Ein neues Diagramm wird erstellt und die Balkendiagrammdaten werden geplottet. Plt.show() zeigt das Diagramm an.



4 Quellen

- https://www.geeksforgeeks.org/a-search-algorithm/
- https://stackoverflow.com/questions/6618515/sorting-list-according-to-corresponding-values-from-a-parallel-list
- https://stackoverflow.com/questions/268272/getting-key-with-maximum-value-in-dictionary
- https://en.wikipedia.org/wiki/Maze_generation_algorithm
- http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html