Aufgabe 1: Stromrallye

Teilnahme-Id: 52493

Bearbeiter/-in dieser Aufgabe:
Christoph Waffler

20. April 2020

Inhaltsverzeichnis

Stromrallye lösen

Im Folgenden Abschnitt wird die Teilaufgabe a) der Aufgabe Stromrallye erläutert.
Die Dokumentation für die Teilaufgabe b) ist in einem separaten Teil vorzufinden.

Lösungsidee

Ziel dieser Aufgabe ist es, den Roboter so durch das Spielbrett zu steuern, dass am Ende alle Batterien leer sind.

Die Lösungsidee kann in folgende Schritte zusammengefasst werden:

1. Initiiere einen leeren Graphen *G*.
   Überprüfe für jede Ersatzbatterie *b* folgendes: Wähle für *b* alle anderen Ersatzbatterien *Z* aus, die von dem Feld von *b* aus mit der Ladung der Batterie *b* erreichbar sind.
   Füge für jede Batterie *z, z ϵ Z,* eine Kante mit *b* als Start- und *z* als Zielknoten zum Graph *G*. Die Gewichtung der Kante ist der Ladungsverbrauch von *b* zu *z*.
   Für die Bordbatterie des Roboters, die sich auf dem Startpunkt des Roboters befindet, wird dasselbe durchgeführt.
2. Wähle denjenigen Pfad im Graphen *G* aus, der der längste im Graphen *G* ist, sodass alle Batterien am Ende ‚leergefahren' sind.
3. Übertrage diesen Pfad in Schrittanweisungen für den Roboter, bspw. ‚Schritt nach oben'.
4. Gebe diese Schrittanweisungen aus.

Im Folgenden werden die einzelnen Schritte genauer erläutert und auftretende Probleme dabei geklärt.

Vorwort zur Abstandsberechnung

Da in der vorliegenden Aufgabe ein Spielbrett verwendet wird, in dem der Roboter nur einen Schritt nach oben, unten, links und rechts betätigen kann, wird zur Berechnung der Distanz zwischen zwei Objekten weitestgehend die Manhattan-Distanz verwendet. Die Manhattan-Distanz zwischen zwei Punkten ist die Summe der Beträge der Differenzen ihrer einzelnen Koordinaten. [1]

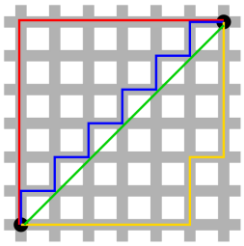


Abbildung 1: *Quelle: Datei: Manhattan distance.svg - https://de.wikipedia.org*

So wird die Manhattan-Distanz zwischen zwei Punkten P(x1/y1) und Q(x2/y2) berechnet:

$$distanz = |x1 - x2| + |y1 - y2|$$

Dabei gibt es mehrere Wege mit derselben Manhattan-Distanz von einem Punkt zu einem anderen zu gelangen. In der Abbildung 1 sind die Linien rot, blau und gelb Beispiele für die Manhattan-Distanz. Die grüne Linie stellt den euklidischen Abstand dar.

Der euklidische Abstand hingegen ist die Länge einer direkten Verbindung zwischen zwei Punkten, welche im zweidimensionalen Raum mithilfe des Satzes von Pythagoras berechnet werden kann:

So wird der euklidische Abstand zwischen zwei Punkten P(x1/y1) und Q(x2/y2) berechnet:

$$distanz = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

[1] *Quelle: Manhattan-Metrik - https://de.wikipedia.org*

Erstellung des Graphen

Als Datenstruktur wird ein gerichteter, gewichteter Graph $G_{Batterien}$ erstellt. Für jede gegebene Ersatzbatterie wird überprüft, ob sie die anderen Ersatzbatterien mit ihrer Ladung erreichen kann. Die eine Ersatzbatterie kann die andere Batterie erreichen, wenn die Manhattan-Distanz zu dieser anderen Batterie kleiner oder gleich der Ladung der Ausgangsbatterie ist.

Zudem soll dazu überprüft werden, ob der Roboter von dieser Batterie aus mit seinen gegebenen Schrittmöglichkeiten (also nach rechts, links, oben oder unten) überhaupt die andere Batterie erreichen kann und nicht der Weg durch eine weitere Batterie verhindert wird.

Ermittlung einer Abfolge von Schritten von einem Start- zu einem Zielpunkt

Um eine Abfolge von Schritten ermitteln zu können – falls diese überhaupt existiert -, wird lokal zur Überprüfung ein zweiter Graph G_{Felder} erstellt, in dem diesmal die Knoten des Graphen durch einzelne Felder des Spielfelds dargestellt werden.
Es wird zuerst der Bereich des Spielfelds eingegrenzt, in dem die Suche des Weges zwischen den beiden Batteriefeldern stattfinden soll. Die Außengrenzen hierfür sind durch die Koordinaten des Start- und Zielpunktes definiert.
Anschaulich wird dies durch Abbildung 2 dargestellt, in der überprüft werden soll, ob ein Weg zwischen der blauen Batterie und der orangenen Batterie existiert. Eine weitere Ersatzbatterie wird durch das rote Viereck dargestellt. Die Außengrenzen des Bereichs sind durch violette Linien dargestellt.

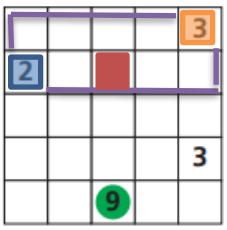


Abbildung 2: modifiziertes Bild aus der Aufgabenstellung

In dem vordefinierten Bereich wird nun für jedes Feld überprüft, ob es seine Nachbarfelder erreichen kann. Dies ist gegeben, wenn sich keine Ersatzbatterie (außer der Start- und Zielbatterie) auf dem Nachbarfeld befindet.
Für jedes Feld werden nun seine freien Nachbarfelder zum lokalen Graphen G_{Felder} hinzugefügt, wobei diesmal die Gewichtung für jede Kante 1 ist, da man eine Ladung verbraucht, um in das Nachbarfeld zu gelangen. Startknoten der Kante ist das Ausgangsfeld und Zielknoten ist das freie Nachbarfeld.

Anschließend wird mithilfe des modifizierten A*-Algorithmus der kürzeste Weg zwischen der Start- und Zielbatterie im lokalen Graphen G_{Felder} ermittelt.
Im Gegensatz zum Dijkstra-Algorithmus wird beim A*-Algorithmus eine Heuristik verwendet, mit welcher eine zielgerichtete Suche ermöglicht und somit die Laufzeit verringert wird. [2]

Der Dijkstra-Algorithmus besitzt eine Komplexität von $O(n^2)$ bei einem Graph mit n Knoten [3], welche vom A*-Algorithmus nur im worst-case erreicht werden könnte [2].

Beim A*-Algorithmus werden immer diejenigen Knoten zuerst untersucht, die wahrscheinlich schnellstmöglich zum Ziel führen. Jedem bekannten Knoten wird dabei ein bestimmter f-Wert zuordnet, mit welchem abgeschätzt werden kann, wie lang der Pfad vom Start- zum Zielknoten über diesen betrachteten Knoten wäre.
Der f-Wert setzt sich aus der Summe des g- und des h-Werts zusammen.
Der g-Wert gibt die bisherigen Kosten vom Startknoten zum aktuellen Knoten an und der h-Wert gibt die mithilfe einer heuristischen Funktion geschätzten Kosten vom aktuell betrachteten Knoten zum Zielknoten an.

Somit gilt für den aktuell betrachteten Knoten k: $f(k) = g(k) + h(k)$

Als heuristische Funktion wird die Manhattan-Distanz zwischen dem aktuellen Knoten und Zielknoten berechnet.

[2] *Quelle: A*-Algorithmus - https://de.wikipedia.org*
[3] *Quelle: Dijkstra-Algorithmus - https://de.wikipedia.org*

Der A*-Algorithmus wählt somit immer den Knoten mit dem besten f-Wert aus, wodurch diese Suche auch als „best first search“ bezeichnet werden kann.[4]

(Eine nähere Beschreibung des A-Algorithmus ist der Umsetzung zu entnehmen.)*

Falls nun die Suche des A*-Algorithmus erfolgreich war und ein Weg zwischen der Start- und Zielbatterie im Graphen G_{Felder} gefunden wurde, ist somit bestätigt worden, dass die andere Ersatzbatterie von der aktuellen Batterie ausgehend erreichbar ist. Dadurch kann diese Ersatzbatterie als Zielknoten zum Graphen $G_{Batterien}$ hinzugefügt werden. Die Gewichtung der Kante ist der Betrag der Ladung, die der Roboter von der Start- zur Zielbatterie theoretisch verbrauchen würde.

Dieselbe Vorgehensweise, wie oben beschrieben, wird für die Bordbatterie des Roboters von seiner Startposition ausgehend verwendet.

Auswahl des längsten Pfades

Nachdem der Graph $G_{Batterien}$ erstellt wurde und alle erreichbaren Ersatzbatterien hinzugefügt wurden, soll nun der längste Pfad in $G_{Batterien}$ ausgewählt werden. Denn nur auf dem längsten Pfad kann sich der Roboter so bewegen, dass am Ende alle Batterien entladen sind.
Aufgrund der Tatsache, dass der Roboter seine Bordbatterie b gegen die Ersatzbatterie e am Boden austauscht, wenn dieser sich auf dem Feld p der Ersatzbatterie e befindet, verändern sich die von p aus erreichbaren anderen Ersatzbatterien bei einer Ladungsdifferenz der Batterien b und e.

Liegt eine solche unterschiedliche Ladung der beiden Batterien vor, so müssen für das Feld p die erreichbaren Ersatzbatterien mit der neuen Ladung b neu ermittelt werden.

Beträgt die Ladung b 0, die auf das Feld p gelegt wird, so können von p aus keine anderen Ersatzbatterien erreicht werden und der zum Feld p dazugehörige Knoten wird aus dem Graphen $G_{Batterien}$ gelöscht.

Mit diesen Bedingungen soll nun mithilfe eines Verfahrens, das an die Tiefensuche angelehnt ist, durch den Graphen $G_{Batterien}$ der längste Pfad gefunden werden.

Folgende Schritte werden dabei ausgeführt:

1. Füge den aktuellen Knoten k zum Pfad f.
2. Ändere die Ladung des Vorgängers v, der sich in f vor k befindet, falls sich seine Ladung verändert hat.
3. Ändere, wenn nötig, die erreichbaren Batterien von v.
4. Abbruch der Tiefensuche, falls alle Batterien leer sind.
5. Rufe für die Nachfolgeknoten N die Tiefensuche rekursiv auf, und übergebe dabei die veränderte Ladung k_{neu} für den Knoten k, die sich durch den Ladungsverbrauch $d_{Verbrauch}$ von k zu n ergibt,
   es gilt: $n \in N$.
   Die neue Ladung k_{neu} für den Knoten k ist somit: $k_{neu} = k_{alt} - d_{Verbrauch}$

[4] *Quelle: Difference and advantages between Dijkstra & A star - https://stackoverflow.com/*

Zu beachten ist dabei, dass die Ladung des Vorgängerknotens *v* erst bei einem zweiten Aufruf der Tiefensuche verändert werden soll.
Dargestellt wird dies durch folgende Abbildungen:

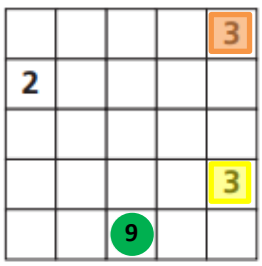


Abbildung 3: Ausgangspunkt Roboter grün markiert, Batterien sind gelb und orange

Pfad: [(3,5)]

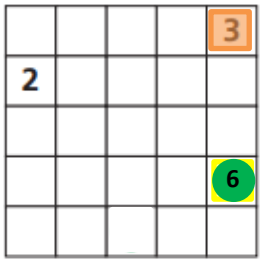


Abbildung 4: Roboter bewegt sich zur gelben Batterie

Pfad: [(3,5), (5, 4)]

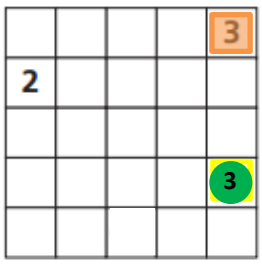


Abbildung 5: Roboter nimmt die Ersatzbatterie auf und legt seine auf das Feld

Pfad: [(3,5), (5,4)]

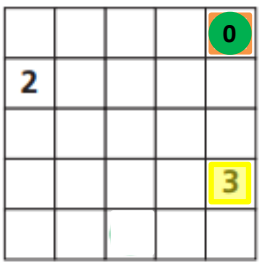


Abbildung 6: Roboter bewegt sich zur orangenen Batterie

Pfad: [(3,5), (5,4), (5,1)]

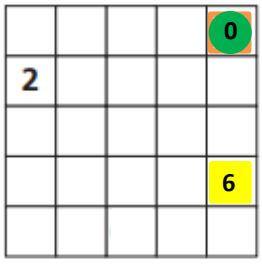


Abbildung 7: Ladung des gelben Feldes wird aktualisiert

Pfad: [(3,5), (5,4), (5,1)]

In Abb. 5 nimmt der Roboter die Ersatzbatterie auf und legt seine Bordbatterie auf das gelbe Feld. Dabei kommt es zu einem Ladungsaustausch aufgrund der unterschiedlichen Ladungen.
Die Ladung dieses Felds kann und wird jedoch erst nach dem 4. Schritt aktualisiert, nachdem der Roboter bereits auf einem nächsten Feld ist. In Abb. 7 erfolgt jetzt dieser Ladungsaustausch des vorherig besuchten Felds.

Zudem sollen bereits besuchte Knoten mehrmals im Pfad auftauchen können, da der Roboter zu bereits besuchten Feldern zurückgehen muss, falls dort eine Ladung > 0 liegt.

Die oben beschrieben 5 Schritte der modifizierten Tiefensuche werden mit den Startknoten k_{Start} für die Ausgangsposition des Roboters zu Beginn des Programms aufgerufen.

Findet die Tiefensuche einen geeigneten Pfad im Graphen $G_{Batterien}$, der alle Batterien auf 0 setzt, so ‚greift' die Abbruchbedingung und der bisherige Pfad wird zurückgegeben.
Nun kann mit der Ermittlung der einzelnen Schritte des Roboters fortgefahren werden, da der gefundene Pfad nur die einzelnen besuchten Felder enthält.

Wird die Abbruchbedingung jedoch in keinem Fall erreicht, also alle möglichen Kombinationen des Weges wurden geprüft, so kann die Tiefensuche keinen Pfad zurückgeben und der Benutzer erhält den Hinweis, dass die eingegebene Spielsituation nicht lösbar ist.

(Eine Optimierung der Tiefensuche ist im Abschnitt Komplexität des Problems beschrieben)

Ermittlung der einzelnen Schritte des Roboters

Ist die Ermittlung des längsten Pfades p_l mithilfe der Tiefensuche erfolgreich verlaufen, so werden nun die einzelnen Schritte des Roboters mithilfe des Pfades p_l berechnet. Der Roboter hat folgende Schrittmöglichkeiten zur Auswahl: Schritt nach oben, nach unten, nach links und nach rechts.

Für alle aufeinanderfolgende Knoten k und k', ($k, k' \in p_l$) wird mithilfe des Verfahrens, das im Abschnitt zur Erstellung des Graphen bereits unter „Ermittlung der Abfolge der Schritte vom Start- zum Zielknoten" erläutert wird, eine Abfolge von Schritten gefunden.
Dieses Mal wird ebenfalls ein lokaler Graph G_{Felder} erzeugt, indem ebenfalls wieder alle für diesen Bereich relevanten Felder als Knoten hinzugefügt werden.

Den Pfad $p_{k,k'}$ zwischen k und k', den der A*-Algorithmus nun gefunden hat, wird noch zu möglichen Schritten des Roboters ‚konvertiert'.
Für alle Punkte in $p_{k,k'}$ wird nun überprüft, ob sich die x- oder y-Koordinaten zwischen dem aktuellen Punkt

P(x1/y1) und seinem Nachfolgepunkt *P'(x2/y2)* positiv oder negativ verändern.
Dafür wird zuerst $\Delta x = x2 - x1$ und $\Delta y = y2 - y1$ berechnet.
Folgende Fälle können dabei eintreten:

- $\Delta x > 0$: Schritt nach rechts
- $\Delta x < 0$: Schritt nach links
- $\Delta y > 0$: Schritt nach unten
- $\Delta y < 0$: Schritt nach oben

Diese Fallunterscheidung wird nun für alle Punkte im Pfad $p_{k,k'}$ durchgeführt und in der Abfolge *a* gespeichert.

Dieses eben genannte Vorgehen für die Knoten *k* und *k'* wird für alle Knoten in p_l ausgeführt und alle Schritte werden in der gesamten Abfolge *A* gespeichert.

Dadurch dass es passieren kann, dass der Roboter zum Schluss von p_l seine leere Batterie abgibt und die letzte Ersatzbatterie mit Ladung > 0 aufnimmt, wird diese im beschriebenen Pfad p_l nicht leergefahren und somit sind noch keine Schritte dafür in *A* gespeichert. Diese müssen ebenfalls noch ermittelt werden.

Dabei wird unterschieden, ob diese letzte Ladung *l*, $l > 1$ oder $l = 1$ ist.

- Ist $l > 1$, so werden zwei freie Nachbarpunkte P_n und P_n' von seinem letzten Standort ausgehend ermittelt, auf denen sich keine leere Ersatzbatterie befindet. Mithilfe von diesen Punkten kann der Roboter seine letzte Ladung leerfahren, indem er zwischen P_n und P_n' solange hin und her fährt, bis $l = 0$ eintritt.

- Ist $l = 1$, so wird sich der Roboter nur noch einen Schritt innerhalb des Spielfelds nach oben bzw. nach unten bewegen.

Ausgabe der Schrittanweisungen

Die berechnete Abfolge der Schritte *A* wird nun noch von programminternen als Zahl repräsentierte Schrittmöglichkeiten, z.B. 0 steht für Schritt nach oben, in deutsche Sprache ‚konvertiert' und in der Abfolge $A_{Deutsch}$ gespeichert.

Die Abfolge $A_{Deutsch}$ wird zum Schluss ausgegeben. Auch eine Visualisierung der Schritte des Roboters in der Umgebung des Spielfelds erfolgt mithilfe eines eigenen GUIs.

Komplexität des Problems

Im Folgenden wird die Komplexität des Problems dargestellt, welches sich aus der Aufgabenstellung ergibt.

Probleme in der Informatik (auch Mathematik) können anhand ihrer Zeitkomplexität in verschiedene Klassen unterteilt werden: [5]

- Klasse P: **deterministisch** in Polynomialzeit lösbare Probleme
  → gewähltes Maschinenmodell der Zeitklasse ist deterministisch und damit realisierbar
  → in Realität: zeitlicher Aufwand für diese Probleme wächst maximal polynomial und sind somit effizient lösbar

- Klasse NP: **nichtdeterministisch** in Polynomialzeit lösbare Probleme
  → gewähltes Maschinenmodell ist nichtdeterministisch und besitzt parallele Berechnungspfade (nicht realisierbar)
  → bekannte deterministische Algorithmen benötigen für diese Probleme einen exponentiellen

[5] *Quelle: Komplexitätstheorie - https://de.wikipedia.org*

Rechenaufwand
→ Es existiert *vermutlich* keine effiziente Lösung der Probleme. [6]

Es wird vorausgesetzt, dass gilt: $P \neq NP$.

Beweis der NP-Vollständigkeit des Problems STROMRALLYE
(angelehnt an „Beispiel: NP-Vollständigkeit zeigen" – Link des KIT: crypto.iti.kit.edu)

Problem der Aufgabenstellung:

Gegeben: Ein gerichteter Graph *G = (V, E)*
Gesucht: Längster Pfad in *G*, mit dem alle Knoten *V* besucht werden.
→ Innerhalb der Tiefensuche meiner Lösungsidee wird bei einem Ladungsaustausch einer ‚neuer' Knoten zum Graphen hinzugefügt, der wieder besucht werden darf.

Es müssen zwei Dinge gezeigt werden:

- STROMRALLYE $\in NP$
- STROMRALLYE ist NP-schwer. Hierfür reduzieren wir das als NP-vollständig vorausgesetzte HAMILTONWEGPROBLEM[7] auf das Problem STROMRALLYE. Wir zeigen also, dass STROMRALLYE mindestens genauso schwer ist wie das HAMILTONWEGPROBLEM.

Zunächst einmal wird deutlich gemacht, dass STROMRALLYE in NP liegt. Das ist der Fall: Wählt man einen beliebigen Pfad für den Graphen der STROMRALLYE, dann ist die Korrektheit des Pfades in $O(|V_{Pfad}|)$ Schritten verifizierbar, indem man simuliert, ob der Roboter bei Befolgen der Knoten V_{Pfad} im geratenen Pfad alle Batterien ‚leerfährt'. Die Überprüfung ob ein Pfad „richtig" ist, ist somit in polynomieller Zeit durchführbar.

STROMRALLYE ist NP-schwer, denn wir können das HAMILTONWEGPROBLEM auf STROMRALLYE reduzieren. Sei *I* eine Instanz des Hamilton-Wegs mit *G = (V, E)*. Anschließend erzeugen wir folgende Instanz *I'* der STROMRALLYE mit *G' = (V', E')*: Wir nehmen an, *G' = G*, und setzen die Anzahl der Kanten *E'* in *G'* auf *E'* = |*V*| - 1.
Nur wenn *G* einen Hamilton-Weg enthält, so existiert ein längster Pfad der Länge *E* in *G'*.

Somit ist das Problem von STROMRALLYE bewiesen NP-schwer und kann daher nicht ohne Weiteres in polynomieller Zeit gelöst werden.

Aufgrund der obengenannten Problematik bezüglich der Zeitkomplexität wird die Tiefensuche der STROMRALLYE mit einer Heuristik optimiert, um die Tiefensuche zu beschleunigen:

Optimierung der Tiefensuche – Heuristik

Spätestens bei Testen der Beispiele ‚stromrallye1.txt' und vor allem ‚stromrallye2.txt' ist die beanspruchte Laufzeit der Tiefensuche ohne Heuristik keineswegs erwartbar.

[6] *Quelle: NP (Komplexitätsklasse) - https://de.wikipedia.org*
[7] *Quelle: Hamiltonkreisproblem - https://de.wikipedia.org*

Die erste Idee einer Heuristik war, immer diejenigen Batterien bei der Tiefensuche als erste zu besuchende Nachbarknoten auszuwählen, die noch eine größere Ladung als die anderen besitzen und sich nahe am Ausgangpunkt befinden. Dadurch sollte sich folgender Pfad für das Beispiel ‚stromrallye2.txt' ergeben.

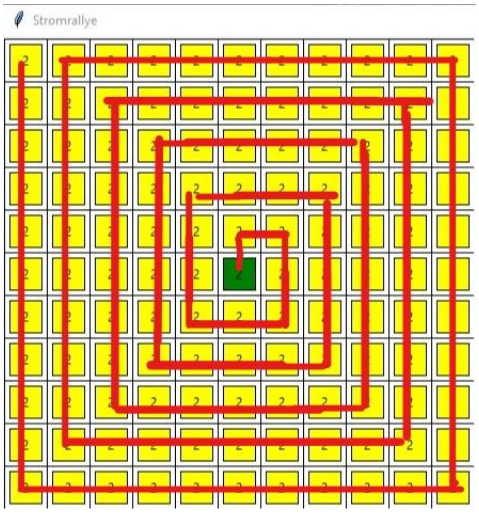


Abbildung 8:
Ersatzbatterien sind gelb dargestellt (jede Ladung=2)
die Startposition des Roboters ist grün

der erste Teil des Wegs ist rot dargestellt

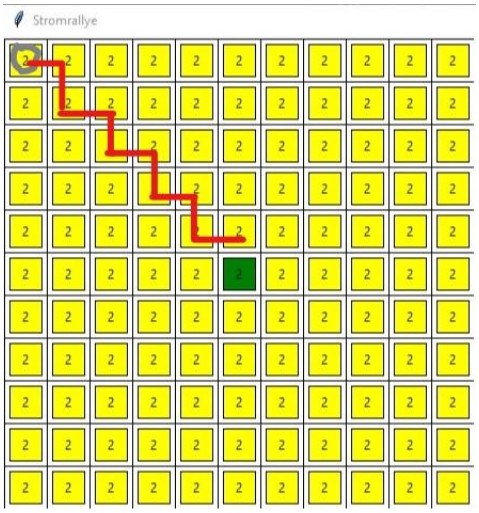


Abbildung 9:
Anfang des 2. Teil des Wegs ist rot dargestellt (vom grauen Feld) aus

Alle Ersatzbatterien haben zu Beginn die Ladung 1

Wie in Abbildung 8 zu sehen ist, würde der Roboter den Weg im Kreis um die Startposition nehmen und im Feld oben links mit dem ersten Teil des Wegs enden, nachdem alle Ersatzbatterien die Ladung = 1 besitzen. Anschließend (alle Ersatzbatterien mit Ladung=1) würde der Roboter vom Feld oben links (in Abb. 9 grau dargestellt) sofort in Richtung der Startposition quer durch das Spielfeld gehen und keine Systematik beachten.

Daher wird die Heuristik hingegen so gewählt, dass diejenigen Ersatzbatterien bei der Tiefensuche als nächste Nachbarknoten ausgewählt werden, die die größte Ladung besitzen und am weitesten vom Startpunkt des Roboters entfernt sind.
Der Abstand zum Startpunkt des Roboters wird mit *d* bezeichnet, die aktuelle Ladung der Ersatzbatterie mit *b*.

Dadurch würde sich der erste ‚Entwurf' einer verbesserten heuristischen Funktion *f* wie folgt ergeben: $f = d * b$

Da jedoch gewünscht ist, dass systematisch zuerst diejenigen Batterien besucht werden, die noch eine höhere Ladung besitzen, sollte die Ladung *b* in der heuristischen Funktion *f* stärker gewichtet werden als der Abstand *d* zum Startpunkt des Roboters.
Daher wird in *f* das Quadrat von *b* verwendet, dies stärker gewichtet wird als der Abstand *d*.
Daraus folgt folgende Formel: $f = d * b^2$

Bei der Auswahl der nächsten Nachbarknoten in der Tiefensuche werden diejenigen Ersatzbatterien priorisiert, die den höchsten Wert der heuristischen Funktion besitzen.

In einem wie in Beispiel ‚stromrallye2.txt' dargestelltem sehr dichten Spielfeld aus Ersatzbatterien bewegt sich der Roboter zu Beginn in Richtung Rand und arbeitet sich dann schrittweise in Richtung Ausgangsposition des Roboters zurück.
Eine genaue Abfolge der Schritte für dieses Beispiel ist im Abschnitt Beispiele nachzulesen.

Besteht das Spielfeld jedoch nicht aus dicht aneinander gereihten Ersatzbatterien, so kann sich dies negativ auf die Laufzeit des Programms auswirken, wie im „Eigenen Beispiel 7" oder Beispiel „stromrallye5.txt" zu sehen ist. Die gewählte Heuristik kann daher nur bestimmte dichte Spielfelder optimieren und kommt bei

Spielsituation die eher „verstreut“ sind und aus höheren Ladungen bestehen an ihre Grenzen. Trotzdem ist sie für viele Fälle ideal wie im Bereich Beispiele anhand der Laufzeiten zu sehen ist.

Umsetzung

Die Lösungsidee wird objektorientiert in Python umgesetzt.
(Anmerkung: Die hier in der Umsetzung verwendete Bezeichnung für Variablen kann von der Bezeichnung der Variablen im Quellcode abweichen.)

Die zwei wichtigsten Klassen sind die Klasse Graph und die Klasse Berechnungen.
In der Klasse Graph wird die Datenstruktur eines Graphs verwaltet und die Klasse Berechnungen ermittelt, welchen Pfad der Roboter zurücklegen muss.
Die Eingabe der Spielsituation und die Visualisierung der Spielumgebung und des laufenden Roboters werden in eigenen Klassen realisiert, die nicht Hauptbestandteil dieser Erläuterung sein sollen.

Datenstruktur Graph

Die Datenstruktur Graph wird in Form einer Adjazenzliste implementiert. Der Graph soll dabei gerichtet und kantengewichtet sein.
Die Adjazenzliste speichert in einem Dictionary alle Startknoten als Key und die Zielknoten mit deren dazugehörigen Gewichtungen in einer Liste als Item des Keys. Die Zielknoten und die Gewichtung der jeweiligen Kante wird dabei ebenfalls in einem eigenen Dictionary gespeichert, indem der Zielknoten als Key fungiert und der Betrag der Gewichtung als dazugehöriges Item.
Die Adjazenzliste könnte zum Beispiel dann so aussehen:

```
{
        (1, 1):   [
                        { (1, 2): 1 },
                        { (2, 1): 1 }
                ]
}
```

Hier stehen {…} für ein Dictionary und […] für eine Liste, (x, y) für die Koordinaten eines Knotens.
Das oben gezeigte Beispiel zeigt, dass von dem Startknoten (1, 1) jeweils eine Kante zu den Zielknoten (1, 2) und (2, 1) verläuft, die jeweils mit 1 gewichtet sind.

Im Folgenden werden die Methoden der Klasse Graph kurz dargestellt.

Mithilfe der Methode *add_Kante(start, ende, gewichtung)* wird zur Adjazenzliste eine neue Kante mit *start* als Startknoten, *ende* als Zielknoten und *gewichtung* als deren Gewichtung hinzugefügt.

Durch die Methode *delete_Kante (start, entfernendes_item)* wird vom Startknoten (*start*) das mit *entfernendes_item* bezeichnete Dictionary entfernt.

Durch die Einführung der Methode *__getitem__(key)* kann mithilfe des „Eckigen-Klammern-Operators“ [] die Adjazenzliste indiziert werden.

Die Methode *aktualisiereNachfolger(knoten, neue_nachfolger)* wird jedes Mal aufgerufen, wenn sich die erreichbaren Batterien eines Felds verändern und somit der Graph aktualisiert werden muss.
Die Liste *neue_nachfolger* enthält alle Zielknoten mit deren Gewichtungen, die dem Startknoten nach der Aktualisierung zugeordnet werden. Dabei werden bisher vorhandene Zielknoten, die nicht in *neue_nachfolger* sind, entfernt.

Dazu wird als erstes eine Kopie $k_{ursprünglich}$ der ursprünglichen Zielknoten mit deren Gewichtungen erstellt. Anschließend wird über die Liste *neue_nachfolger* iteriert, und alle neuen Zielknoten erstmal mit der Methode *add_Kante(...)* hinzugefügt. Ist ein Zielknoten bereits vor Aktualisierung ein Zielknoten des Startknoten, so soll

dieser auch nach der Aktualisierung erhalten bleiben und wird von $k_{ursprünglich}$ entfernt.
Jene Zielknoten die nach der Iteration über der Liste *neue_nachfolger* noch in $k_{ursprünglich}$ vorhanden sind, werden nun durch die Methode *delete_Kante(...)* entfernt, da diese nicht mehr vorhanden sein sollen.

Klasse Berechnungen

Die Klasse Berechnungen ist jene Klasse, in der alles Grundlegende bezüglich der Aufgabenstellung ermittelt wird.
Einige Methoden und Verfahren werden im Folgenden erklärt, da die Main-Methode sich auf diese aufbaut.

Initialisierung

Zu Beginn wird die Klasse Berechnung vom Eingabefenster aufgerufen und erhält eine Liste mit Strings, welche die Eingabe enthalten (z.B. ['5', '3,5,9', '3', '5,1,3', '1,2,2', '5,4,3']).
Das erste Element der Liste ist die Größe des Spielfelds und wird dem Klassenattribut *self.size* als Datentyp *int* zugeordnet. Danach folgen die Startkoordinaten und die Anfangsladung des Roboters, welche mithilfe einer eigenen Methode *zuPunkt(eingabe)* von bspw. '3,5,9' zu *x=3, y=5* und *ladung=9* konvertiert wird. Diese Werte werden dem Klassenattribut *self.roboter* zugewiesen.

Die darauffolgende Zahl gibt die Anzahl der Ersatzbatterien an und wird dem Attribut *self.batterien* zugewiesen. Für die darauffolgenden String-Elemente der Ersatzbatterien, bspw. '5,1,3', '1,2,2' , wird ebenfalls die Methode *zuPunkt(eingabe)* aufgerufen und deren ‚konvertierte' Elemente in der Liste *self.batterien* gespeichert.

In der Initialisierungsmethode *__init__(...)* wird am Schluss die main-Methode aufgerufen.

Methoden zur Abstandsbestimmung

Wie bereits in der Lösungsidee beschrieben werden zwei verschiedene Arten der Abstandermittlung benötigt.
Für den euklidischen Abstand wird in der Methode *euklidischerAbstand(x1, y1, x2, y2)* zuerst Δx und Δy bestimmt. Von den beiden Differenzen wird das Quadrat gebildet. Aus dessen Summe wird die Wurzel gezogen, welche den endgültigen Betrag der Distanz angibt. Diese Distanz ist der Rückgabewert der Methode.

In der Methode *manhattanDistanz(x1, y1, x2, y2)* für die Berechnung der Manhattan-Distanz werden am Anfang die Beträge von Δx und Δy bestimmt. Die Distanz ist die Summe der beiden Beträge, welche den Rückgabewert dieser Methode darstellt.

Methode zur Bestimmung freier Nachbarpunkte

Die Methode *findeFreieNachbarpunkte(x_akt, y_akt, potenzielle_hindernisse)* ermittelt freie Nachbarpunkte.
Die Koordinaten *x_akt* und *y_akt* geben dabei das Ausgangsfeld an und die Liste *potenzielle_hindernisse* speichert die x- und y-Koordinaten derjenigen Felder, auf denen Ersatzbatterien sind.

Nun soll ermittelt werden, welche Nachbarfelder des Ausgangsfelds frei sind, d.h. dass dort keine Ersatzbatterien sind.
Überprüft werden dabei das Feld oberhalb, unterhalb, rechts und links davon, falls diese vorhanden sind. Da Felder an den Rändern bzw. in den Ecken des Spielfelds nicht alle Nachbarfelder haben können, soll die Überprüfung durch eine Reihe von if-Bedingungen gestützt werden, welche die vorliegende Position des Felds eingrenzen.

Zu Beginn wird überprüft, ob sich das Ausgangsfeld in *potenzielle_hindernisse* befindet. Ist dies der Fall so wird nichts zurückgegeben. Wenn nicht, dann wird mit der Ermittlung fortgesetzt.
In der Liste *nachbarn* werden alle erreichbaren Nachbarpunkte gespeichert.
Für die benachbarten Felder *unten*, *links*, *rechts* und *oben* werden als erstes die Koordinaten ermittelt. Wenn sich die benachbarten Felder in *potenzielle_hindernisse* befinden, so wird den jeweiligen Variablen *None* zugewiesen.
Mit einer Reihe von if- Bedingungen werden folgende Fälle unterschieden:

- y_akt ist größer als 1:
  → aktuelle Position ist am oberen Rand
  Zur Liste *nachbarn* wird das Feld *oben* hinzugefügt.

- *y_akt* ist kleiner als *self.size*:
  → aktuelle Position ist nicht am unteren Rand
  Zur Liste *nachbarn* wird das Feld *unten* hinzugefügt.

- *x_akt* ist größer als 1:
  → aktuelle Position ist nicht am linken Rand
  Zur Liste *nachbarn* wird das Feld *links* hinzugefügt.

- *x_akt* ist kleiner als *self.size*:
  → aktuelle Position ist nicht am rechten Rand
  Zur Liste *nachbarn* wird das Feld *rechts* hinzugefügt.

(Es wird angenommen, dass sich die aktuelle Position im Spielfeld befindet.)

Die Liste *nachbarn* ist der Rückgabewert der Methode *findeFreieNachbarpunkte(...)*.

A*-Algorithmus

Die Methode *astar(start, ziel, graph)* soll im gegebenen Graph *graph* vom Startknoten *start* zum Zielknoten *ziel* den kürzesten Weg berechnen.
Dabei wird eine Schätzfunktion verwendet, um das Verfahren zu optimieren, indem der Abstand vom aktuellen zum Zielknoten geschätzt wird. Als Schätzfunktion *h* wird die Methode *heurstik_astar(...)* verwendet, welche die Manhattan-Distanz für die beiden Knoten berechnet. Wie bereits in der Lösungsidee erwähnt, setzt sich der *f*-Wert aus $f = g + h$ zusammen.

Zu Beginn der Methode *astar* werden zwei leere Dictionary f_{Dict} und g_{Dict} erzeugt, in denen die *f*- und *g*-Werte für die besuchten Knoten gespeichert werden. Der *g*-Wert für den Startknoten *start* ist zu Beginn 0 und der *f*-Wert von *start* besteht daher nur noch aus dem heuristischen Wert, der mithilfe der Methode *heuristik_astar(...)* berechnet wird.

Noch offene und bereits geschlossene Knoten werden in jeweils einem Set *offene_knoten* und *geschlossene_knoten* gespeichert. Zu Beginn wird der Startknoten *start* zu *offene_knoten* hinzugefügt.
Der Verlauf des Wegs kann - falls der Zielknoten erreicht wird - mithilfe des Dictionary *gekommen_von* zurückverfolgt werden.

Mit einer while-Schleife werden folgende Anweisungen solange ausgeführt, bis keine Knoten mehr in *offene_knoten* sind:

- Wähle mithilfe einer for-Schleife denjenigen Knoten von *offene_knoten* aus, der den geringsten *f*-Wert besitzt. Dies wird mithilfe von f_{Dict} realisiert. Dieser Knoten ist nun der aktuelle Knoten *k*.

- Abbruchbedingung der while-Schleife: Ist der aktuelle Knoten *k* gleich dem Zielknoten *ziel*, so wird der Pfad rückwärtsgegangen, indem mit einer while-Schleife über das Dictionary *gekommen_von* iteriert wird und die jeweiligen Vorgänger zur Liste *p* hinzugefügt werden.
  Die Liste *p* wird umgekehrt, damit der Zielknoten an letzter und der Startknoten an erster Stelle ist.
  Die Liste *p* wird mit der Länge *l* des kürzesten Wegs *p* zurückgegeben. Die Länge *l* ist der *f*-Wert des Zielknoten *ziel* (hier identisch mit dem aktuellen Knoten *k*).

- Ist die Abbruchbedingung nicht eingetreten, so markiere den aktuellen Knoten *k* als geschlossen, dabei wird *k* von *offene_knoten* entfernt und zu *geschlossene_knoten* hinzugefügt.

- Aktualisiere anschließend die *g*-Werte für die Nachbarknoten *N*, welche die Nachfolgeknoten des aktuellen Knoten *k* sind.

Mithilfe einer for-Schleife wird für jeden Nachbarknoten *n, n ϵ N*, zuerst überprüft, ob dieser bereits ausgeschöpft ist und daher *n* in *geschlossene_knoten* ist.
Ist dies der Fall, so wird durch *‚continue‘* mit dem nächsten Nachbarknoten die for-Schleife fortgesetzt.

Wenn nicht, so wird für *n* ein neuer *g*-Wert berechnet, der mit *g_wert_kandidat* bezeichnet wird.
Ist *n* noch nicht in *offene_knoten* gespeichert, so wird er zu *offene_knoten* hinzugefügt.

Anschließend wird überprüft, ob der neue Wert *g_wert_kandidat* schlechter (=größer) ist, als der vorher gefundene.
Ist dies der Fall, so wird durch *‚continue‘* mit dem nächsten Element die for-Schleife fortgesetzt, da der bessere *g*-Wert behalten werden sollte.

Wenn nicht, so wird der *g*-Wert von *n* im Dictionary g_{Dict} auf den Wert *g_wert_kandidat* gesetzt.
Der Vorgänger von *n* wird im Dictionary *gekommen_von* auf den aktuellen Knoten gesetzt.

Nun wird der Abstand zum Zielknoten mithilfe der Methode *heuristik_astar(...)* geschätzt.
Der neue *f*-Wert von *n* wird in f_{Dict} aktualisiert, der sich aus der Summe des eben berechneten heuristischen Werts und des *g*-Werts in g_{Dict} ergibt.

Endet die while-Schleife ohne die Abbruchbedingung zu erreichen, so wurde kein Weg gefunden und es wird *None* zurückgegeben.

Methode zur Konvertierung eines Pfads bestehend aus Punkten in Schrittanweisungen

Mithilfe der Methode *findeWeg(x_start, y_start, x_ziel, y_ziel)* wird die Abfolge der Schritte des Roboters vom Startpunkt(*x_start, y_start)* zum Zielpunkt(*x_ziel, y_ziel*) ermittelt.
Die Schritte werden im Programm als Zahl dargestellt:

- 0: Schritt nach oben
- 1: Schritt nach unten
- 2: Schritt nach links
- 3: Schritt nach rechts

Zu Beginn werden die Differenzen $\Delta x = x_{ziel} - x_{start}$ und $\Delta y = y_{ziel} - y_{start}$ zur Eingrenzung des Suchbereichs berechnet.
Anschließend wird eine neue Liste *batterien_x_y* erstellt, in der die x- und y-Koordinaten der Batterien von *self.batterien* gespeichert werden. Die jeweiligen Ladungen werden nicht benötigt.

Da der Start- und Zielpunkt nicht als potenzielles Hindernis gespeichert werden sollen, werden diese – falls sie in der Liste *batterien_x_y* sind – von *batterien_x_y* entfernt.
Sonst würde der Algorithmus keinen Weg finden, wenn beispielsweise der Zielpunkt in *batterien_x_y* vorhanden ist.

Der lokalen Variablen *graph* wird ein neuer, leerer Graph zugewiesen und die aktuelle Position auf *x_start* und *y_start* gesetzt.

Mithilfe der Variable *step_*x und *step_y* kann gespeichert werden, in welche Richtung über den Bereich mithilfe der for-Schleife iteriert werden muss.
Dabei soll *step_x* bzw. *step_y* bei einem positiven Delta (Δx bzw. Δy) +1 sein, bei negativem Delta -1.

Es folgen zwei ineinander verschachtelte for-Schleifen:
Mithilfe der ersten for-Schleife wird nun die Variable *i* von *y_start* bis einschließlich *y_ziel* mit *step_y*

Schrittrichtung iteriert.
In dieser ersten for-Schleife wird der y-Koordinate der aktuellen Position der aktuelle Wert *i* zugewiesen.

Danach folgt in dieser for-Schleife eine weitere for-Schleife, in der nun die Variable *j* von *x_start* bis einschließlich *x_ziel* mit *step_x* Schrittrichtung iertiert wird.
Hier wird der x-Koordinate der aktuellen Position der aktuelle Wert *j* zugeordnet.

Von dieser aktuellen Position ausgehend werden jetzt die Nachbarfelder mithilfe der Methode *findeFreieNachbarn(*aktuelle_position, batterien_x_y)* ermittelt.
Für jedes existierende freie Nachbarfeld wird zu *graph* eine neue Kante mit der aktuellen Position als Startpunkt und dem freien Nachbarfeld als Zielpunkt mit Gewichtung=1 hinzugefügt.

Sind beide genannten for-Schleifen fertig, so wird mithilfe des A*-Algorithmus der kürzeste Weg in *graph* von dem Punkt S (*x_start, y_start*) zu Z (*x_ziel, y_ziel*) berechnet.

Falls die aufgerufene Methode *astar(...)* einen Weg *weg* gefunden hat, so kann dieser in eine Abfolge von Schritten umgewandelt werden.
Diese Abfolge von Schritten sollen in der Liste *abfolge_schritte* gespeichert werden.

Für jedem Punkt *P* in *weg* (außer dem ersten Punkt) wird ein Teil-Weg konstruiert, welcher genau einen Schritt des Roboters beschreibt:

- Von dem Punkt *P* und seinem Vorgänger *P'* werden die Differenzen der x- und y-Koordinaten berechnet: $\Delta x = x_{P'} - x_P$ und $\Delta y = y_{P'} - y_P$
- Je nach Differenz der Koordinaten muss eine bestimmte Bewegung ausgeführt werden, um von *P'* zu *P* zu gelangen:
  - $\Delta x > 0: bewegung = 3$
  - $\Delta x < 0: bewegung = 2$
  - $\Delta y > 0: bewegung = 1$
  - $\Delta y < 0: bewegung = 0$
- Die ermittelte Variable *bewegung* wird zur Liste *abfolge_schritte* hinzugefügt.

Am Ende wird die Liste *abfolge_schritte* zurückgegeben.

Hat der aufgerufene A*-Algorithmus vorhin jedoch keinen Weg gefunden, so wird *None* zurückgegeben.

Methode zur Berechnung der erreichbaren Batterien

In der Methode *erreichbareBatterien(x_start, y_start, ladung, restliche_batterien)* werden mithilfe der Manhattan-Distanz alle Batterien ermittelt, die von dem gegebenen Feld aus mit der gegebenen Ladung erreichbar sind.

Der Betrag der Manhattan-Distanz wird über eine eigene Methode berechnet. Dieser Betrag gibt Auskunft darüber, wie viel Ladung vom Ausgangspunkt (*x_start, y_start*) zur jeweiligen Ersatzbatterie benötigt wird.

Falls die gegebene Ladung > 0 ist, werden folgende zwei Filterungen der Liste *restliche_batterien* unternommen:

- Filtere die gegebene Liste *restliche_batterien* so, dass nur jene Batterien *b, b ϵ restliche_batterien*, erhalten bleiben, deren Manhattan-Distanz zum Punkt S (*x_start, y_start*) kleiner oder gleich *ladung* ist. Diese gefilterte Liste wird *erstauswahl* genannt.
- Filtere die eben erstelle Liste *erstauswahl* so, dass nur noch jene Batterien *b*, *b ϵ erstauswahl*, erhalten bleiben, bei denen die Methode *findeWeg(x_start, y_start, *b)* einen Weg (nicht *None*) zurückgibt. Diese gefilterte Liste wird *erreichbare_batterien* genannt.

Die Liste *erreichbare_batterien* ist der Rückgabewert dieser Methode.

Ist die der Methode als Parameter übergebene Ladung *ladung* = 0, so wird *None* zurückgegeben, da keine anderen Batterien erreicht werden können.

Verfahren der Tiefensuche

Bemerkung: Die Methode *heuristik_dfs(x, y, batterien_aktuelle_ladung)* berechnet zuerst den euklidischen Abstand *s* vom Punkt *S(x, y)* zum Startpunkt des Roboters. Dann wird die aktuelle Ladung der Batterie ausgelesen, indem das gegebene Dictionary *batterien_aktuelle_ladung* für *(x, y)* verwendet wird.
Das Produkt aus dem Abstand *s* und dem Quadrat der aktuellen Ladung wird zurückgegeben und stellt somit den heuristischen Wert dar, mit dem die Nachbarknoten verglichen werden können.

Die Methode *dfs(graph, aktueller_knoten, alte_ladung, a_alte_ladung, aktuelle_ladung_batterien, pfad=[])* ist die Implementierung einer rekursiven Tiefensuche im gegebenen Graph. *Dfs* steht für die englische Bezeichnung für die Tiefensuche: depth-first search, kurz DFS

Kurze Erklärung der Parameter der Methode:

- *graph*: Datenstruktur Graph zum Speichern der aktuellen Kantenbeziehungen der Ersatzbatterien
- *aktueller_knoten*: aktuell ausgewählter Knoten, der zum Pfad hinzugefügt wird
- *alte_ladung*: Ladung des vorherigen Knotens, die als *a_alte_ladung* im nächsten Schritt weitergegeben wird
- *a_alte_ladung*: veränderte Ladung der vorvorherig besuchten Batterie, dessen Ladung in *aktuelle_ladung_batterien* nun aktualisiert werden sollte
- *aktuelle_ladung_batterien*: Dictionary zum Speichern der aktuellen Ladungen der Batterien
- *pfad*: Liste zum Speichern des aktuellen Wegs (mit Knoten des Graphs)

Zu Beginn der Methode *dfs* wird der aktuell übergebene Knoten *aktueller_knoten* zu *pfad* hinzugefügt.

- Besitzt *pfad* aktuell eine Länge von 2, so hat sich der Roboter von seinem Startpunkt auf das Feld einer Ersatzbatterie bewegt und auf seinem Startpunkt befindet sich nun keine ‚Ladung' mehr, daher wird der Startpunkt des Roboters aus dem Dictionary *aktuelle_ladung_batterien* vollständig gelöscht.
  Anschließend wird die Liste *restliche_batterien* erstellt, die sich aus allen x- und y-Koordinaten derjenigen Batterien aus dem Dictionary *aktuelle_ladung_batterien* ergeben, deren Ladung > 0 ist.

- Falls *pfad* jedoch eine Länge > 2 besitzt, so wird immer die Ladung des Vorvorgängers aktualisiert.
  Dazu wird das vorletzte Element *alt_knoten* aus *pfad* ausgewählt. *A_alte_ladung* beschreibt den Betrag der Ladung, welchen *alt_knoten* nun besitzen soll. Ist seine bisherige Ladung in *aktuelle_ladung_batterien* nicht gleich seinem neuen Wert *a_alte_ladung*, so muss als erstes das Dictionary *aktuelle_ladung_batterien* für *alt_knoten* den Wert *a_alte_ladung* erhalten.
  Anschließend wird die Liste *restliche_batterien* ermittelt, die sich aus allen x- und y-Koordinaten derjenigen Batterien aus dem Dictionary *aktuelle_ladung_batterien* ergeben, deren Ladung > 0 ist.

  Dann werden die neuen erreichbaren Batterien *neue_erreichbare_batterien* mithilfe der Methode *erreichbareBatterien(...)* von dem Feld des *alt_knoten* ausgehend berechnet.
  Die Liste *neue_erreichbare_batterien* wird mithilfe der built-in Funktion *map()* so konvertiert, dass die Manhattan-Distanz zwischen der jeweiligen Batterie und *alt_knoten* zusammen mit den dazugehörigen x- und y-Koordinaten gespeichert wird.

  Anschließend wird mithilfe der Methode *aktualisiereNachfolger(...)* des Objekts *graph* die Nachfolger von *alt_knoten* im Graph *graph* aktualisiert.

- Falls *pfad* jedoch eine Länge < 2 besitzt, so wird nur die Liste *restliche_batterien* ermittelt, die sich aus allen x- und y-Koordinaten derjenigen Batterien aus dem Dictionary *aktuelle_ladung_batterien* ergeben, deren Ladung > 0 ist.

Nach den genannten if-Bedingungen wird die aktuelle Ladung *aktuelle_ladung* von *aktueller_knoten* im Dictionary *aktuelle_ladung_batterien* ausgelesen.

Anschließend folgt die Abbruchbedingung der rekursiven Tiefensuche:
Hier wird mithilfe einer if-Bedingung überprüft, ob nur ein Element in *restliche_batterien* ist und *aktueller_knoten* dieses letzte Element in *restliche_batterien* ist und *alte_ladung* gleich 0 ist.
Alte_ladung muss gleich 0 sein, da sonst im nächsten Schritt eine Ersatzbatterie immer noch eine Ladung > 0 hat. Wird die Abbruchbedingung erreicht, so wird *pfad* zusammen mit *aktuelle_ladung* zurückgegeben.

Tritt die Abbruchbedingung nicht ein, so werden danach die Nachfolgerknoten von *aktueller_knoten* nach ihrem heuristischen Wert der Methode *heuristik_dfs(...)* absteigend in der Liste *sortierte_nachfolger* sortiert gespeichert.
Diese Sortierung der Liste der Nachfolgerknoten ist die Optimierung der Tiefensuche, damit sich die Laufzeit des Programms verringert.

Anschließend wird für jeden Nachbarknoten *n* in *sortierte_nachfolger* die Methode *dfs* rekursiv aufgerufen. Dabei verändern sich die übergebenen Parameter:

- Der Graph *graph* wird mithilfe des Moduls *copy* tiefenkopiert, da sonst eine Veränderung von *graph* in einem Nachbarknoten, den Graph der anderen Nachbarknoten auch beeinflussen würden, was nicht erwünscht ist.
- Der Parameter *alte_ladung* erhält die Differenz aus *aktuelle_ladung* und die Gewichtung *kanten_gewicht* der Kante von *aktueller_knoten* zu *n*.
  Somit folgt für *alte_ladung*: $alte\_ladung = aktuelle\_ladung - kanten\_gewicht$
- Der Parameter *a_alte_ladung* wird mit dem Wert von *alte_ladung* aufgerufen.
- Das Dictionary *aktuelle_ladung_batterien* wird mit *.copy()* ebenfalls kopiert, da sonst dieselben Probleme beim rekursiven Aufruf auftreten würden (wie bei *graph*).
- Der Pfad wird ebenfalls mit *.copy()* kopiert und weiter übergeben.

Daraus ergibt sich folgender Methodenaufruf für den Nachfolgeknoten *n*:

dfs(graph=copy.deepcopy(graph), aktueller_knoten=n, alte_ladung=(aktuelle_ladung - kanten_gewicht), a_alte_ladung=alte_ladung, aktuelle_ladung_batterien=aktuelle_ladung_batterien.copy(), pfad=pfad.copy())

Main-Methode

Die Methode *main()* wird bei der Initialisierung der Klasse *Berechnungen* aufgerufen und steuert sozusagen die gesamten Berechnungen.
Am Anfang wird ein Objekt der Klasse Graph *self.graph* erzeugt.

Mit einer for-Schleife wird für jede Ersatzbatterie *b* in *self.batterien* die erreichbaren Ersatzbatterien *E* durch die Methode *erreichbareBatterien(...)* ermittelt. Jene erreichbaren Batterien werden in Form einer Kante *k* anschließend zum Graphen hinzugefügt. Die Kante *k* ist aufgebaut aus dem Startknoten *b* und Zielknoten *e*, $e \in E$, mit der Gewichtung der Manhattan-Distanz, die durch die Methode *manhattanDistanz(...)* berechnet wird.

Für den Startpunkt des Roboters wird ebenfalls mit der Methode *erreichbareBatterien(...)* die erreichbaren Batterien ermittelt und wie auf dieselbe Weise wie eben genannt zum Graphen *self.graph* hinzugefügt.

Anschließend wird ein Dictionary *aktuelle_ladung_batterien* zum Speichern der aktuellen Ladungen der Batterien erstellt. Für jede Ersatzbatterie wird die x- und y-Koordinate in Form eines *Tuple* als Key und die Ladung als dazugehöriges Item gespeichert.
Für den Roboter mit seiner Anfangsladung der Bordbatterie wird dasselbe durchgeführt.

Mithilfe der Methode *dfs(...)* wird ermittelt, in welcher Reihenfolge der Roboter die Felder ansteuern muss, damit alle Batterien am Ende leer sind.
Die Tiefensuche *dfs* liefert, falls erfolgreich, ein *item* zurück, das den *pfad* und die *restliche_ladung* beinhaltet. Falls kein *item* zurückgegeben wird, so wird ausgegeben, dass die eingelesene Spielsituation nicht lösbar ist und kein Pfad gefunden wurde.

Falls ein *item* existiert, so wird dies getrennt als *pfad* und *restliche_ladung* gespeichert.

Pfad soll nun in eine Abfolge von Schritten (*abfolge_schritte*) umgewandelt werden.
Dazu wird für jeden Punkt P zusammen mit seinem Vorgänger P‘ in *pfad* mit der Methode *findeWeg(*P‘, *P)* die Schritte *s* ermittelt, welche der Roboter befolgen muss. Die Liste *abfolge_schritte* wird mit der Liste *s* erweitert.

Anschließend wird aus der Liste *self.batterien* eine neue Liste *batterien_x_y* erstellt, in der nur noch die x- und y-Koordinaten der Batterien von *self.batterien* gespeichert werden.

Jetzt fehlen noch jene Schritte, die der Roboter gehen muss, damit *restliche_ladung* gleich 0 ist.

Ist *restliche_ladung* gleich 1, so wird zu *abfolge_schritte* noch ein Schritt nach oben bzw. ein Schritt nach unten hinzugefügt (je nachdem welcher Schritt aufgrund der Position des letzten besuchten Felds von *pfad* möglich ist).

Ist *restliche_ladung > 1*, so wird logischerweise mehr als ein Schritt benötigt, um die *restliche_ladung* zu entladen.
Das letzte in *pfad* besuchte Feld wird der Variablen *letzte_position* zugewiesen. Von *letzte_position* ausgehend wird mit der Methode *findeFreieNachbarpunkte(...)* eine Liste *nachbarn1* mit besuchbaren Nachbarpunkten erstellt.
Anschließend wird geprüft, ob ein Tuple in der Liste enthalten ist, d.h. dass es darin mindestens ein freies Nachbarfeld gibt.
Ist dies der Fall, so wird vom ersten Element *erster_nachbar* der Liste *nachbarn1* erneut die Methode *findeFreieNachbarpunkte(...)* aufgerufen und eine Liste *nachbarn2* mit von *erster_nachbar* aus besuchbaren Nachbarfeldern erstellt.

Es wird erneut geprüft, ob ein Tuple, also die Koordinaten eines freien Nachbarn in *nachbarn2* sind.
Ist dies der Fall, so wird das erste Element von *nachbarn2* der Variable *zweiter_nachbar* zugewiesen.

Jetzt wird noch der Schritt für jeweils den Weg von *letzte_position* zu *erster_nachbar*, von *erster_nachbar* zu *zweiter_nachbar* und von *zweiter_nachbar* zu *erster_nachbar* mithilfe der Methode *findeWeg(...)* ermittelt.

Zu *abfolge_schritte* wird der Schritt von *letzte_position* zu *erster_nachbar* hinzugefügt und *restliche_ladung* um 1 verringert.

Nun wird solange *restliche_ladung > 0* ist, von *erster_nachbar* zu *zweiter_nachbar* und umgekehrt hin und her gegangen. Dabei wird immer der jeweilige Schritt zu *abfolge_schritte* hinzugefügt und *restliche_ladung* um 1 verringert.

Die *abfolge_schritte* ist nun vollständig ermittelt und gibt den gesamten Weg an, den der Roboter zurücklegen muss, damit alle Batterien leer sind.
Jedoch sind die Schritte noch in Form von Zahlen gespeichert, welche jetzt zum Schluss der Methode *main()* noch mithilfe einer for-Schleife und if-Bedingungen zu deutschen Wörtern konvertiert werden und in *abfolge_schritte_deutsch* gespeichert werden:

- 0 → ‚oben‘
- 1 → ‚unten‘
- 2 → ‚links‘
- 3 → ‚rechts‘

Die Liste *abfolge_schritte_deutsch* wird in der Konsole ausgegeben.

Eine Visualisierung der Spielumgebung mit Ausführung der Schritte des Roboters ist durch den Aufruf entsprechender Methoden einer eigenen Klasse gegeben (Name der Klasse im Quellcode: *Environment*).

Die jeweilige Laufzeit einer Eingabe wird ebenfalls ausgegeben. Dazu wird die Zeit bei Aufruf der Main-Methode gestartet und vor Ausgabe *abfolge_schritte_deutsch* gestoppt. Die Differenz ist die benötigte Zeit.

Beispiele

Der Roboter wird meinem GUI in einem grünen Quadrat dargestellt, die Ersatzbatterien in einem gelben. Die aktuellen Ladungen der Batterien sind in Form einer Zahl in der Mitte der jeweiligen Quadrate zusehen.

Beispiel ‚stromrallye0.txt' der BwInf-Website

>> Pfad gefunden > [(3, 5), (5, 4), (5, 1), (5, 4), (1, 2)]
>> Abfolge von 17 Schritten für den Roboter in deutscher Sprache:
> ['rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'unten', 'unten', 'unten', 'links', 'oben', 'links', 'links', 'links', 'oben', 'oben', 'rechts']
>> Laufzeit des Programms: 0.0019948482513427734 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Beispiel ‚stromrallye1.txt' der BwInf-Website

>> Pfad gefunden > [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (2, 10), (3, 10), (4, 10), (5, 10), (6, 10), (7, 10), (8, 10), (9, 10), (10, 10), (10, 9), (10, 8), (10, 7), (10, 6), (10, 5), (10, 4), (10, 3), (10, 2), (10, 1), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (8, 9), (7, 9), (6, 9), (5, 9), (4, 9), (3, 9), (2, 9), (2, 8), (3, 8), (4, 8), (5, 8), (6, 8), (7, 8), (8, 8), (8, 7), (8, 6), (8, 5), (8, 4), (8, 3), (8, 2), (8, 1), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (6, 7), (5, 7), (4, 7), (3, 7), (2, 7), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (6, 1), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (4, 5), (3, 5), (2, 5), (2, 4), (3, 4), (4, 4), (4, 3), (4, 2), (4, 1), (3, 1), (3,
2), (3, 3), (2, 3), (2, 2), (2, 1)]
>> Abfolge von 100 Schritten für den Roboter in deutscher Sprache:
> ['unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben',
'oben', 'oben', 'oben', 'oben', 'links', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'oben', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'links', 'unten', 'unten', 'links', 'oben', 'oben', 'unten']
>> Laufzeit des Programms: 0.17499017715454102 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Beispiel ‚stromrallye2.txt' der BwInf-Website

>> Pfad gefunden > [(6, 6), (5, 6), (4, 6), (3, 6), (2, 6), (1, 6), (1, 5), (1, 4), (1, 3), (1, 2), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (11, 1), (11, 2), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (10, 11), (9, 11), (8, 11), (7, 11), (6, 11), (5, 11), (4, 11), (3, 11), (2, 11), (1, 11), (1, 10), (1, 9), (1, 8), (1, 7), (2, 7), (2, 8), (2, 9), (2, 10), (3, 10), (4, 10), (5, 10), (6, 10), (7, 10), (8, 10), (9, 10), (10, 10), (10, 9), (10, 8), (10, 7), (10, 6), (10, 5), (10, 4), (10, 3), (10, 2), (9, 2), (8, 2), (7, 2), (6, 2), (5, 2), (4, 2), (3, 2), (2, 2), (2, 3), (2, 4), (2, 5), (3, 5), (3, 4), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (8, 9), (7, 9), (6, 9), (5, 9), (4, 9), (3, 9), (3, 8), (3, 7), (4, 7), (4, 8), (5, 8), (6, 8), (7, 8), (8, 8), (8, 7), (8, 6), (8, 5), (8, 4), (7, 4), (6, 4), (5, 4), (4, 4), (4, 5), (5, 5), (6, 5), (7, 5), (7, 6), (7, 7), (6, 7), (5, 7), (4, 7), (3, 7), (2, 7), (1, 7), (1, 8), (1, 9), (1, 10), (1, 11), (2, 11), (3, 11), (4, 11), (5, 11), (6, 11), (7, 11), (8, 11), (9, 11), (10, 11), (11, 11), (11, 10), (11, 9), (11, 8), (11, 7), (11, 6), (11, 5), (11, 4), (11, 3), (11, 2), (11, 1), (10, 1), (9, 1), (8, 1), (7, 1), (6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 6), (2, 5), (2, 4), (2, 3), (2, 2), (3, 2),

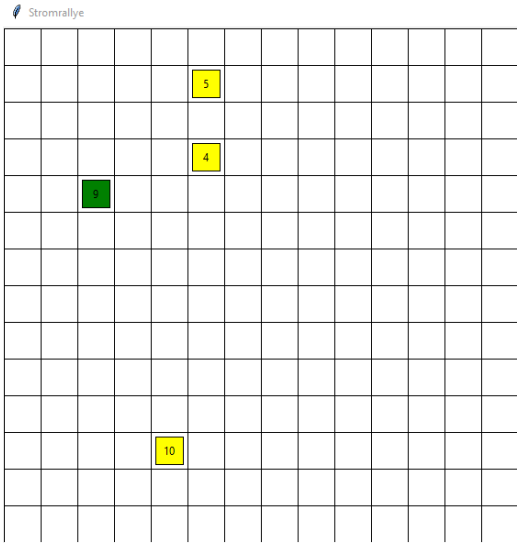
(4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 2), (10, 2), (10, 3), (10, 4), (10, 5), (10, 6), (10, 7), (10, 8), (10, 9), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (5, 10), (4, 10), (3, 10), (2, 10), (2, 9), (2, 8), (3, 8), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9), (9, 8), (9, 7), (9, 6), (9, 5), (9, 4), (9, 3), (8, 3), (7, 3), (6, 3), (5, 3), (4, 3), (3, 3), (3, 4), (3, 5), (3, 6), (4, 6), (4, 5), (4, 4), (5, 4), (6, 4), (7, 4), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (7, 8), (6, 8), (5, 8), (4, 8), (4, 7), (5, 7), (5, 6), (5, 5), (6, 5), (7, 5), (7, 6), (7, 7), (6, 7)]
>> Abfolge von 242 Schritten für den Roboter in deutscher Sprache:
> ['links', 'links', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'rechts', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'links', 'oben', 'oben', 'rechts', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'links', 'links', 'unten', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'unten', 'unten', 'rechts', 'oben', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'links', 'oben', 'oben', 'rechts', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'links', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'links', 'links', 'oben', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'unten', 'unten', 'links', 'oben']
>> Laufzeit des Programms: 0.7359671592712402 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Beispiel ‚stromrallye3.txt'

>> Die eingegebene Spielsituation ist nicht lösbar!
Kein Pfad wurde gefunden!

>> Laufzeit des Programms: 0.0019998550415039062 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Anmerkung dieses Beispiel ist nicht lösbar:
Der Roboter muss als erstes zur unteren Ersatzbatterie, da diese die Ladung=10 besitzt und sonst nicht mehr zu dieser gelangen könnte, würde er zuerst zu einer anderen Ersatzbatterie fahren.
Von dieser unteren Ersatzbatterie benötigt er genau 9 Schritte um zur Ersatzbatterie mit Ladung=4 zu gelangen. Er würde 11 benötigen, um seine Ladung zu entleeren, was jedoch nicht geht, da keine negativen Ladungen existieren dürfen.
Daher ist dieses Beispiel unter diesen Voraussetzungen nicht lösbar.



Ausgangssituation des Beispiels ‚stromrallye3.txt'

Beispiel ‚stromrallye4.txt'

>> Pfad gefunden > [(40, 25)]

>> Abfolge von 20 Schritten für den Roboter in deutscher Sprache:
> ['links', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links']

>> Laufzeit des Programms: 0.0009953975677490234 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Beispiel ‚stromrallye5.txt'

>> Die eingegebene Spielsituation ist nicht lösbar!

Kein Pfad wurde gefunden!

>> Laufzeit des Programms: 3877.47683429718 Sekunden

(Start der Zeitmessung bei Aufruf der main-Methode)

Dies Spielumgebung ist wie Beispiel ‚stromrallye3.txt' nicht lösbar. Da mein Algorithmus (die Tiefensuche) im worst-case alle möglichen Wege ermittelt, ist die Laufzeit im worst-case exponentiell zur Eingabe.

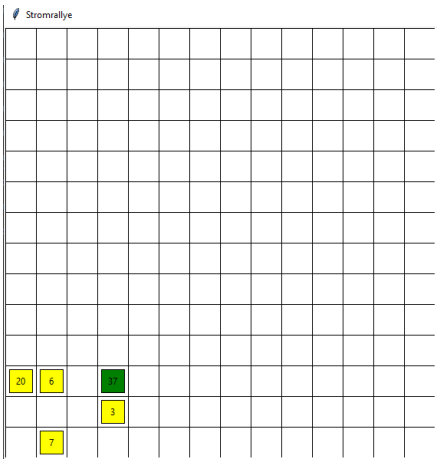
Da hier alle möglichen Wege ausprobiert werden, bevor erkannt wird, dass die Spielsituation nicht lösbar ist, ist die Laufzeit hier nicht ideal.

Testweise wurde das Programm trotzdem laufen gelassen und kommt nach ca. einer Stunde auf das Ergebnis, das die Spielsituation nicht lösbar ist.

Eigenes Beispiel 1:

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***leicht*** *und* ***14x14 Spielfeld***

Eingabe:
14
4,12,37
4
2,14,7
2,12,6
4,13,3
1,12,20



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:
>> Pfad gefunden > [(4, 12), (2, 14), (1, 12), (2, 14), (2, 12), (2, 14), (2, 12), (1, 12), (2, 12), (1, 12), (2, 14), (2, 12), (1, 12), (2, 14), (1, 12), (2, 12), (1, 12), (2, 14), (4, 13), (2, 14), (4, 13), (1, 12), (2, 12), (1, 12), (4, 13)]

>> Abfolge von 73 Schritten für den Roboter in deutscher Sprache:
> ['links', 'unten', 'links', 'unten', 'oben', 'links', 'oben', 'unten', 'rechts', 'unten', 'oben', 'oben', 'unten', 'unten', 'oben', 'oben', 'links', 'rechts', 'links', 'unten', 'rechts', 'unten', 'oben', 'oben', 'links', 'unten', 'rechts', 'unten', 'oben', 'links', 'oben', 'rechts', 'links', 'unten', 'rechts', 'unten', 'oben', 'rechts', 'rechts', 'links', 'links', 'unten', 'oben', 'rechts', 'rechts', 'links', 'links', 'links', 'oben', 'rechts', 'links', 'unten', 'rechts', 'rechts', 'rechts', 'links', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links', 'rechts', 'links']

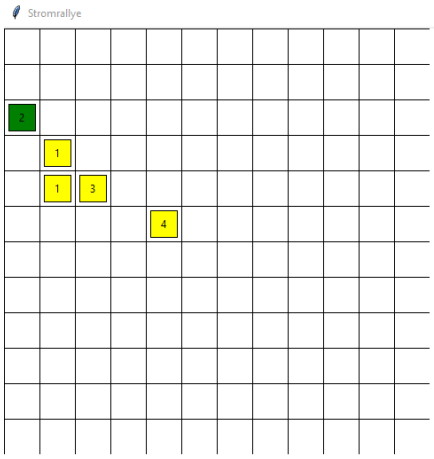
>> Laufzeit des Programms: 0.06978416442871094 Sekunden

Eigenes Beispiel 2:

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***leicht*** *und* ***12x12 Spielfeld***

Eingabe:

12
1,3,2
4
5,6,4
3,5,3
2,5,1
2,4,1



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(1, 3), (2, 4), (2, 5), (3, 5), (5, 6)]

>> Abfolge von 11 Schritten für den Roboter in deutscher Sprache:
> ['rechts', 'unten', 'unten', 'rechts', 'rechts', 'unten', 'rechts', 'links', 'links', 'rechts', 'links']

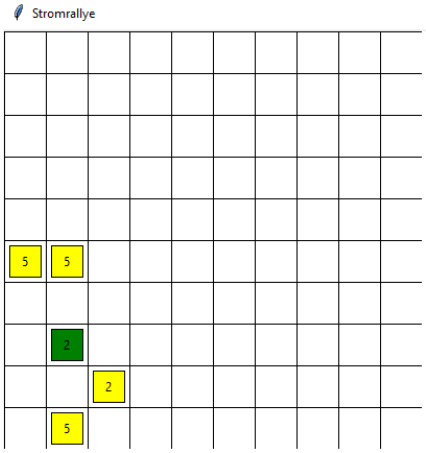
>> Laufzeit des Programms: 0.006994962692260742 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Eigenes Beispiel 3

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***leicht*** *und* ***10x10 Spielfeld***

Eingabe:

10
2,8,2
4
2,6,5
1,6,5
2,10,5
3,9,2



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(2, 8), (2, 6), (1, 6), (3, 9), (2, 10), (1, 6)]

>> Abfolge von 19 Schritten für den Roboter in deutscher Sprache:
> ['oben', 'oben', 'links', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'links', 'unten', 'links', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'unten', 'oben']

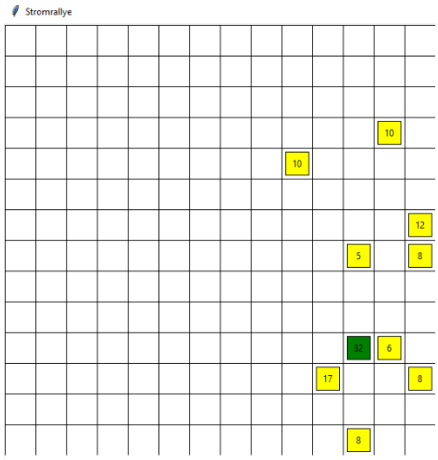
>> Laufzeit des Programms: 0.010021448135375977 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Eigenes Beispiel 4:

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***mittel*** *und* ***14x14 Spielfeld***

Eingabe:

14
12,11,32
9
13,4,10
14,7,12
10,5,10
12,8,5
14,8,8
13,11,6
12,14,8
14,12,8
11,12,17



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(12, 11), (13, 4), (14, 7), (13, 4), (10, 5), (13, 4), (10, 5), (11, 12), (13, 4), (14, 8), (12, 14), (11, 12), (14, 7), (10, 5), (13, 4), (12, 8), (11, 12), (14, 12), (14, 8), (14, 7), (14, 8), (14, 12), (13, 11), (14, 8), (12, 8), (14, 8), (12, 8)]

>> Abfolge von 116 Schritten für den Roboter in deutscher Sprache:
> ['oben', 'oben', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'rechts', 'unten', 'unten', 'unten', 'links', 'oben', 'oben', 'oben', 'unten', 'links', 'links', 'links', 'rechts', 'rechts', 'rechts', 'oben', 'unten', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'unten', 'unten', 'rechts', 'unten', 'unten', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'oben', 'unten', 'unten', 'unten', 'unten', 'rechts', 'unten', 'links', 'links', 'unten', 'unten', 'unten', 'unten', 'unten', 'oben', 'oben', 'links', 'oben', 'oben', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'links', 'oben', 'links', 'oben', 'links', 'links', 'rechts', 'rechts', 'rechts', 'oben', 'unten', 'unten', 'links', 'unten', 'unten', 'links', 'unten', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'unten', 'unten', 'unten', 'unten', 'unten', 'oben', 'links', 'rechts', 'oben', 'oben', 'oben', 'links', 'links', 'rechts', 'rechts', 'links', 'links', 'links']

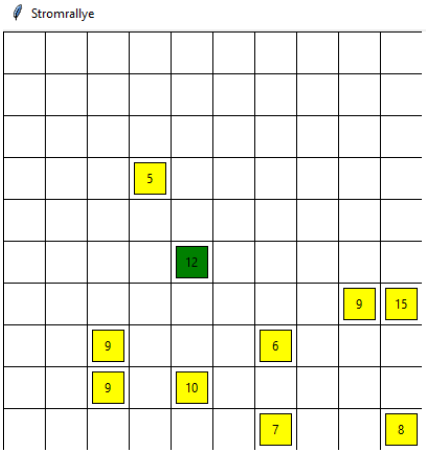
>> Laufzeit des Programms: 98.16500067710876 Sekunden

Eigenes Beispiel 5:

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***mittel*** *und* ***12x12 Spielfeld***

Eingabe:

10
5,6,12
9
3,9,9
3,8,9
4,4,5
5,9,10
7,10,7
9,7,9
10,10,8
10,7,15
7,8,6



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(5, 6), (10, 7), (10, 10), (9, 7), (10, 10), (5, 9), (3, 9), (3, 8), (3, 9), (7, 10), (3, 9), (3, 8), (10, 7), (10, 10), (7, 8), (5, 9), (9, 7), (10, 10), (7, 10), (5, 9), (3, 9), (3, 8), (3, 9), (3, 8), (3, 9), (4, 4)]

>> Abfolge von 92 Schritten für den Roboter in deutscher Sprache:
> ['rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'links', 'oben', 'oben', 'oben', 'unten', 'unten', 'unten', 'rechts', 'links', 'oben', 'links', 'links', 'links', 'links', 'links', 'links', 'oben', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'links', 'links', 'links', 'oben', 'links', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'rechts', 'rechts', 'oben', 'unten', 'unten', 'unten', 'links', 'oben', 'links', 'oben', 'links', 'links', 'unten', 'links', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'rechts', 'links', 'links', 'links', 'links', 'oben', 'links', 'links', 'links', 'oben', 'unten', 'oben', 'unten', 'rechts', 'oben', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'rechts', 'links', 'rechts']

>> Laufzeit des Programms: 0.13999366760253906 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

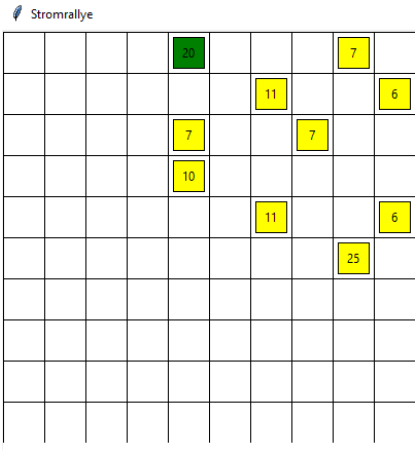
Eigenes Beispiel 6:

Eingabe (im ‚BwInf-Format')

Erzeugt durch mein Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***mittel*** *und* ***10x10 Spielfeld***

Eingabe:

10
5,1,20
9
10,2,6
8,3,7
10,5,6
7,5,11
7,2,11
9,1,7
9,6,25
5,4,10
5,3,7



Visualisierung der eigens erzeugten Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(5, 1), (9, 6), (7, 5), (9, 6), (7, 5), (9, 6), (5, 4), (9, 6), (7, 5), (7, 2), (7, 5), (10, 5), (7, 5), (10, 5), (9, 1), (10, 2), (9, 1), (8, 3), (10, 5), (10, 2), (9, 6), (8, 3), (7, 2), (5, 4), (5, 3), (5, 4), (5, 3), (5, 4), (10, 5), (7, 5), (7, 2), (10, 2), (9, 1)]

>> Abfolge von 110 Schritten für den Roboter in deutscher Sprache:
> ['rechts', 'rechts', 'rechts', 'unten', 'rechts', 'unten', 'unten', 'unten', 'unten', 'links', 'links', 'oben', 'unten', 'rechts', 'rechts', 'links', 'links', 'oben', 'unten', 'rechts', 'rechts', 'oben', 'oben', 'links', 'links', 'links', 'links', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'links', 'links', 'links', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'links', 'oben', 'oben', 'unten', 'rechts', 'links', 'oben', 'unten', 'links', 'unten', 'rechts', 'unten', 'rechts', 'unten', 'oben', 'oben', 'oben', 'links', 'unten', 'unten', 'unten', 'unten', 'oben', 'oben', 'oben', 'links', 'links', 'oben', 'unten', 'unten', 'links', 'links', 'oben', 'unten', 'oben', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'rechts', 'unten', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'links', 'oben', 'links', 'links', 'rechts', 'links']

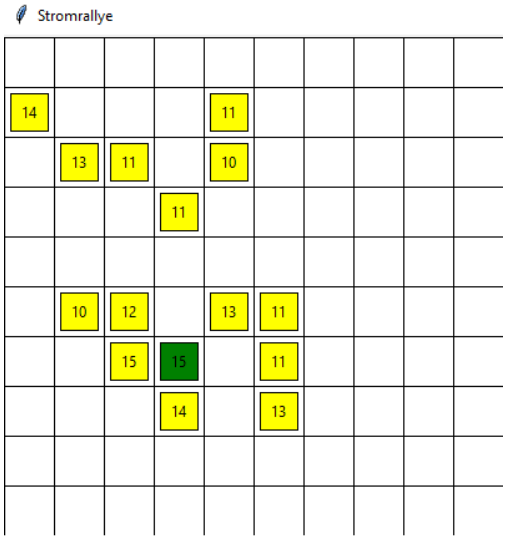
>> Laufzeit des Programms: 0.25499987602233887 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Eigenes Beispiel 7:

Erzeugt mit Programm der Teilaufgabe b) mit Schwierigkeitsgrad: ***schwer*** *und* ***10x10 Spielfeld***

Eingabe:

10
4,7,15
14
4,4,11
5,3,10
5,2,11
1,2,14
3,3,11
2,3,13
6,8,13
6,7,11
6,6,11
4,8,14
5,6,13
3,6,12
3,7,15
2,6,10



eigens erzeugte Spielsituation (Ausgangssituation)

Ausgabe:

>> Pfad gefunden > [(4, 7), (1, 2), (2, 3), (5, 2), (2, 3), (3, 3), (5, 2), (3, 3), (5, 3), (4, 4), (6, 8), (1, 2), (5, 3), (5, 2), (6, 6), (5, 2), (2, 3), (5, 6), (6, 7), (3, 7), (6, 7), (3, 6), (6, 7), (4, 8), (2, 6), (4, 4), (2, 6), (3, 3), (5, 2), (3, 3), (4, 4), (6, 7), (4, 8), (2, 3), (3, 6), (3, 7), (6, 8), (3, 7), (2, 6), (2, 3), (1, 2), (2, 3), (3, 3), (5, 3), (5, 2), (5, 3), (5, 2), (5, 3), (4, 4), (2, 6), (4, 8), (6, 7), (6, 8), (6, 7), (6, 6), (5, 6), (6, 6), (5, 6), (6, 7), (3, 7)]

>> Abfolge von 184 Schritten für den Roboter in deutscher Sprache:
> ['oben', 'oben', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'unten', 'rechts', 'oben', 'rechts', 'rechts', 'rechts', 'links', 'links', 'links', 'unten', 'rechts', 'oben', 'rechts', 'rechts', 'links', 'links', 'unten', 'rechts', 'rechts', 'unten', 'links', 'unten', 'unten', 'unten', 'rechts', 'unten', 'rechts', 'links', 'oben', 'links', 'oben', 'oben', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'rechts', 'unten', 'rechts', 'oben', 'rechts', 'unten', 'unten', 'unten', 'unten', 'oben', 'oben', 'oben', 'oben', 'links', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'unten', 'rechts', 'unten', 'rechts', 'links', 'links', 'links', 'rechts', 'rechts', 'rechts', 'links', 'links', 'oben', 'links', 'rechts', 'unten', 'rechts', 'rechts', 'links', 'links', 'unten', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'rechts', 'rechts', 'links', 'links', 'unten', 'unten', 'oben', 'oben', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'links', 'links', 'unten', 'unten', 'rechts', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'links', 'links', 'unten', 'oben', 'oben', 'oben', 'links', 'links', 'oben', 'oben', 'unten', 'unten', 'rechts', 'unten', 'unten', 'rechts', 'rechts', 'unten', 'rechts', 'links', 'oben', 'links', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'links', 'oben', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'unten', 'oben', 'unten', 'unten', 'links', 'links', 'links', 'unten', 'unten', 'unten', 'unten', 'rechts', 'rechts', 'oben', 'rechts', 'rechts', 'unten', 'oben', 'oben', 'links', 'rechts', 'links', 'unten', 'rechts', 'links', 'links', 'links', 'links']

>> Laufzeit des Programms: 4.686039209365845 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

(Die Laufzeit ist bei diesem Schwierigkeitsgrad zwar deutlich höher als bei den (meisten) Beispiele auf der BwInf-Website aber trotzdem noch deutlich schneller als ein menschlicher Spieler.)

Quellcode

Die Kommentare unter dem Methodenkopf (""" … """) wurden aus Platzgründen etwas gekürzt.

A*-Algorithmus

```
def heuristik_astar(self, knoten1, knoten2):
    """ Methode als heuristische Funktion im A*-Algorithmus
    Es wird die Manhattan-Distanz zwischen zwei Knoten berechnet.

    (weitere Kommentare aus Platzgründen weggelassen …)
    """
    return self.manhattanDistanz(*knoten1, *knoten2)

def astar(self, start, ziel, graph):
    """ A*-Algorithmus zur Berechnung des kürzesten Weg.

    Im gegebenen Graph soll vom Start- zum Zielknoten der kürzeste Weg berechnet w
erden.
    Dabei wird eine Schätzfunktion verwendet, um das Verfahren zu optimieren, inde
m der Abstand zum Zielknoten berechnet wird.
    Als Schätzfunktion wird die Methode heuristik_astar verwendet, welche die Manh
attan-Distanz berechnet.

    (…)
    """
    # Tatsächliche Kosten zu jedem Knoten vom Startknoten aus
    G = {}

    # Geschätzte Kosten vom Start zum Ende über die Knoten
    F = {}

    # Initialisierung der Startwerte
    G[start] = 0
    F[start] = self.heuristik_astar(start, ziel)

    # offene und geschlossene Knoten werden in einem Set gespeichert
    geschlossene_knoten = set()
    # zu Beginn wird der Startknoten zu den offenen Knoten hinzugefügt
    offene_knoten = set([start])
    # Verlauf des Wegs wir in gekommen_von gespeichert
    gekommen_von = {}

    # Solange noch ein Knoten offen ist:
    while len(offene_knoten) > 0:

        # Wähle den Knoten von der offenen Liste aus, der den geringsten F-
Wert besitzen
        aktueller_knoten = None
        aktueller_F_wert = None

        for knoten in offene_knoten:
            if aktueller_knoten is None or F[knoten] < aktueller_F_wert:
                aktueller_F_wert = F[knoten]
                aktueller_knoten = knoten

        # Überprüfe, ob der Zielknoten erreicht wurde
        if aktueller_knoten == ziel:

            # falls ja, wird die Route rückwärts gegangen
```

```
            pfad = [aktueller_knoten]
            while aktueller_knoten in gekommen_von:
                aktueller_knoten = gekommen_von[aktueller_knoten]
                pfad.append(aktueller_knoten)

            # Pfad wird umgekehrt
            pfad.reverse()

            # der Pfad wird mit der Länge zurückgegeben
            return pfad, F[ziel]  # Fertig!

        # Markiere den aktuellen Knoten als geschlossen
        offene_knoten.remove(aktueller_knoten)
        geschlossene_knoten.add(aktueller_knoten)

        # Aktualisierung der Werte für die Nachbarknoten neben dem aktuellen Knote
n
        for item in graph[aktueller_knoten]:
            nachbar_knoten, gewichtung = item[0]

            if nachbar_knoten in geschlossene_knoten:
                # dieser Knoten wurde bereits ausgeschöpft
                continue

            g_wert_kandidat = G[aktueller_knoten] + gewichtung

            # falls der Nachbarknoten noch nicht offen ist,
            # wird er als offener gespeichert
            if nachbar_knoten not in offene_knoten:
                offene_knoten.add(nachbar_knoten)

            elif g_wert_kandidat >= G[nachbar_knoten]:
                # Wenn der G-Wert schlechter als der vorher gefundene ist
                continue

            # G-Wert wird angepasst
            gekommen_von[nachbar_knoten] = aktueller_knoten
            G[nachbar_knoten] = g_wert_kandidat

            # Abstand zum Zielknoten wird geschätzt
            H = self.heuristik_astar(nachbar_knoten, ziel)
            F[nachbar_knoten] = G[nachbar_knoten] + H

    # Falls kein Weg gefunden wurde wird None zurückgegeben
    return None
```

Methode zur Konvertierung eines Pfads bestehend aus Punkten in Schrittanweisungen

```
def findeWeg(self, x_start, y_start, x_ziel, y_ziel):
    """ Methode zum Ermitteln der Abfolge der Schritte vom Start- zum Zielpunkt.

    Die Datenstruktur Graph wird eingesetzt.
    Es wird zuerst der Bereich des Spielfelds eingrenzt, in dem Suche stattfinden
soll.
    Dies Außengrenzen des Bereich sind durch die Koordinaten des Start- und Zielpu
nkts definiert.

    (…)
```

```
    """
    # Die Differenzen der x- und y-
Koordinaten werden zur Eingrenzung des Bereichs berechnet
    delta_x = x_ziel - x_start
    delta_y = y_ziel - y_start

    # nur die x- und y-Koordinaten aller Batterien werden benötigt
    batterien_x_y = [batterie[0:2] for batterie in self.batterien]

    # Start- und Zielpunkt sollen im nachfolgenden Algorithmus nicht als potenziel
les Hindernis gespeichert werden
    # da der Algorithmus sonst keinen Weg finden würde, wenn beispielsweise der Zi
elpunkt in der Liste der potenziellen Hindernisse gespeichert wird
    startpunkt, zielpunkt = (x_start, y_start), (x_ziel, y_ziel)
    if startpunkt in batterien_x_y:
        batterien_x_y.remove(startpunkt)
    if zielpunkt in batterien_x_y:
        batterien_x_y.remove(zielpunkt)

    graph = Graph()

    aktuelle_position = [x_start, y_start]

    # step gibt die Richtung an, in der über den Bereich iteriert wird.
    # Der Bereich ist durch die Koordinaten des Start- und Zielpunkts definiert.

    # ein step ist entweder bei positivem Delta +1, bei negativem Delta -1
    # bei +1 wird von links nach rechts iteriert, bei -1 umgekehrt
    if delta_y > 0:
        step_y = 1
    else:
        step_y = -1

    if delta_x > 0:
        step_x = 1
    else:
        step_x = -1


    # for-
Schleife iteriert von y_start bis einschließliche y_ziel, mit step_y als Schritt (
entweder +1 oder -1)
    for i in range(y_start, y_ziel + step_y, step_y):
        aktuelle_position[1] = i

        # analog zur for-Schleife für die y-Koordinate, diesmal in x-Richtung
        for j in range(x_start, x_ziel + step_x, step_x):
            aktuelle_position[0] = j

            nachbarn = self.findeFreieNachbarpunkte(*aktuelle_position, batterien_
x_y)

            for punkt in nachbarn:
                # falls punkt nicht NoneType ist
                if punkt:
                    graph.add_Kante(
                        tuple(aktuelle_position), punkt, 1)

    return_item = self.astar((x_start, y_start), (x_ziel, y_ziel), graph)
```

```
    # falls überhaupt ein Weg gefunden wurde
    if return_item:
        kürzester_weg, länge = return_item
        abfolge_schritten = []

        for index in range(len(kürzester_weg)):
            if index > 0:
                # Teilweg aus den Punkten P1 und P2
                p1 = kürzester_weg[index-1]
                p2 = kürzester_weg[index]

                # die Differenzen der x- und y-Koordinaten wird berechnet
                delta_x = p2[0] - p1[0]
                delta_y = p2[1] - p1[1]

                # je nach Differenz der Koordinaten muss eine bestimmte Bewegung a
usgeführt werden (z. B. nach rechts gehen),
                # um von P1 zu P2 zu gelangen

                # nach rechts
                if delta_x > 0:
                    bewegung = 3

                # nach links
                elif delta_x < 0:
                    bewegung = 2

                # nach unten
                elif delta_y > 0:
                    bewegung = 1

                # nach oben
                elif delta_y < 0:
                    bewegung = 0

                abfolge_schritten.append(bewegung)

        return abfolge_schritten
    else:
        return None
```

Methode zur Berechnung der erreichbaren Batterien

```
def erreichbareBatterien(self, x_start: int, y_start: int, ladung: int, restliche_
batterien: list):
    """ Mithilfe der Manhattan-Distanz werden alle Batterie ermittelt,
        die von dem gegebenen Punkt aus erreichbar sind.

        Der Betrag der Manhattan-Distanz wird über eine eigene Methode berechnet.
        Dieser gibt in dieser Aufgabe Auskunft darüber, wie viel Ladung zur Ersatz
batterie benötigt wird

        (…)
    """

    # Die gegebene Ladung ist der maximal mögliche Betrag der Manhattan-Distanz
    # daher werden alle Batterien herausgefiltert, dessen Manhattan-
Distanz größer als die gegebene Ladung ist.
```

```
    # Zudem muss die Distanz größer als 0 sein, damit der Ausgangspunkt sich nicht
 selbst als erreichbare Batterie sehen kann
    if ladung > 0:
        erstauswahl = list(filter(
            lambda batterie_item: self.manhattanDistanz(x_start, y_start, batterie
_item[0], batterie_item[1]) <= ladung
            and self.manhattanDistanz(x_start, y_start, batterie_item[0], batterie
_item[1]) > 0, restliche_batterien
        ))


        # Falls für den Weg von dem gegebenen Startpunkt zur Batterie in der Liste
 erstauswahl keine Schritte über Felder existieren,
        # ist die Batterie nicht erreichbar und wird von der Liste entfernt.
        erreichbare_batterien = list(filter(
            lambda batterie_item: self.findeWeg(x_start, y_start, *batterie_item[:
2]) != None, erstauswahl
        ))

        # eine Liste mit erreichbaren Batterien wird zurückgegeben
        return erreichbare_batterien

    # Falls die Ladung <= 0 ist, wird eine leere Liste zurückgegeben, da keine Bat
terien erreicht werden können
    else:
        return []
```

Methoden für das Verfahren der Tiefensuche

```
def filterListeBatterien(self, aktuelle_ladung_batterien: dict):
    """ Als Liste der restlichen Batterien werden nur die x- und y-
Koordinaten aller Batterien benötigt,
        die aktuell eine Ladung > 0 besitzten

        Args:
            aktuelle_ladung_batterien: Dictionary mit den aktuellen Ladungen der B
atterien

        Returns:
            list. Eine Liste mit den x- und y-
Koordinaten der Batterien, die eine Ladung > 0 besitzen.
    """
    # 1. Filtern der Batterien mit Ladung > 0
    restliche_batterien = list(filter(
        lambda batterie: batterie[1] > 0, list(
            aktuelle_ladung_batterien.items())
    ))
    # 2. Nur die x- und y-Koordinaten werden benötigt
    restliche_batterien = list(
        map(lambda batterie: batterie[0], restliche_batterien))

    return restliche_batterien

def heuristik_dfs(self, x: int, y: int, batterien_aktuelle_ladung: dict, graph):
    """ Diese Methode dient als Optimierung der Tiefensuche (depth-first-
search, DFS).

    Diese Methode liefert einen Wert, der die Auswahl zwischen den Nachbarknoten e
ines Knoten eines Graphen optimieren soll.
```

```
    (weitere Kommentare aus platztechnischen Gründen entfernt …)
    """

    # x- und y-Koordinate des Startpunktes des Roboters werden benötigt
    roboter = self.roboter[:2]

    # euklidischer Abstand wird verwendet
    abstand = self.euklidischerAbstand(x, y, *roboter)

    # aktuelle Ladung der Batterie wird ausgelesen
    aktuelle_ladung_batterie = batterien_aktuelle_ladung[(x, y)]

    # das Produkt aus dem Abstand und dem Quadrat der Ladung wird zurückgegeben
    return aktuelle_ladung_batterie**2 * abstand

def dfs(self, graph, aktueller_knoten, alte_ladung, a_alte_ladung, aktuelle_ladung
_batterien, pfad=[]):
    """ Methode zur rekursiven Tiefensuche im gegebenen Graphen

    Args:
        graph (Graph): Graph als Datenstruktur zum Speichern der aktuellen Kantenb
eziehungen der Ersatzbatterien
        aktueller_knoten (tuple): aktuell ausgewählter Knoten, der zum Pfad hinzug
efügt wird
        alte_ladung (int): Ladung des vorherigen Knotens, die als a_alte_ladung im
 nächsten Schritt weitergegeben wird
        a_alte_ladung (int): veränderte Ladung der vorvorherig besuchten Batterie,
 dessen Ladung nun aktualisiert wird
        aktuelle_ladung_batterien (dict): Dictionary zum Speichern der aktuellen L
adungen der Batterien
        pfad (list): Liste zum Speichern der besuchten Batterien (Knoten des Graph
en)

    Returns:
        list. Pfad mit einem Weg, den der Roboter zurücklegen muss, damit alle Bat
terien leer sind
    """
    # aktueller Knoten wird zum Pfad hinzugefügt
    pfad.append(aktueller_knoten)
    #print(f"> Pfad: {pfad}")

    # ist der Pfad länger als 2 Elemente, so wird immer die Ladung des Vorvorgänge
rs aktualisiert
    if len(pfad) > 2:

        # die Ladung des vorvorletzten Elements muss aktualisiert werden
        alter_knoten = pfad[-2]

        # bisherige Ladung des alten Knotens wird ermittelt
        bisherige_ladung = aktuelle_ladung_batterien[alter_knoten]

        # nur wenn die 'neue' (a_alte_ladung) des alten Knotens sich von der bishe
rigen Ladung unterscheidet,
        # ändern sich auch die erreichbaren anderen Ersatzbatterien von der aktuel
len Position ausgehend
        # und somit auch die Nachbarknoten im Graphen
        if a_alte_ladung != bisherige_ladung:
```

```
            # Aktualisierung des Dictionary zum Speichern der aktuellen Ladungen
            # die Ladung des vorvorletzten Elements wird auf a_alte_ladung gesetzt
            aktuelle_ladung_batterien[alter_knoten] = a_alte_ladung

            # restliche Batterien werden ermittelt
            restliche_batterien = self.filterListeBatterien(aktuelle_ladung_batter
ien)

            # aufgrund der veränderten Ladung des Vorvorgängers werden nun seine e
rreichbaren Batterien neu ermittelt
            erreichbare_batterien_neu = self.erreichbareBatterien(*alter_knoten, a
_alte_ladung, restliche_batterien)

            # die Manhattan-Distanz wird berechnet und mithilfe der map()-
Funktion als tuple mit der jeweiligen erreichbaren Batterie gespeichert
            erreichbare_batterien_neu = list(map(
                lambda batterie: (*batterie, self.manhattanDistanz(*batterie, *alt
er_knoten)),
                erreichbare_batterien_neu
            ))

            # die Nachbarknoten der Vorvorgängers werden im Graphen aktualisiert
            graph.aktualisiereNachfolger(alter_knoten, erreichbare_batterien_neu)

        else:
            # restliche Batterien werden ermittelt
            restliche_batterien = self.filterListeBatterien(aktuelle_ladung_batter
ien)

    else:
        # Da der Roboter sich am Anfang bewegt und sozusagen keine Ersatzbatterie
'hinterlässt',
        # wird die Postion aus dem Dictionary gelöscht,
        # Auf dem Startfeld des Roboters befindet sich nun keine Batterie mehr.
        if len(pfad) == 2:
            del aktuelle_ladung_batterien[(self.roboter[0], self.roboter[1])]

        # restliche Batterien werden ermittelt
        restliche_batterien = self.filterListeBatterien(aktuelle_ladung_batterien)

    # die akutelle Ladung des aktuellen Knotens wird ausgelesen
    aktuelle_ladung = aktuelle_ladung_batterien[aktueller_knoten]


    # Falls nur noch eine Batterie übrig ist (die Batterie des aktuellen Knoten),
    # so ist die suche beendet.
    # Der Pfad und die aktuelle Ladung des aktuellen Knotens werden zurückgegeben
    if len(restliche_batterien) == 1 and aktueller_knoten == restliche_batterien[0
] and alte_ladung == 0:
        return pfad, aktuelle_ladung


    # Heuristik
    # Mithilfe der Methode heuristik_dfs() wird für jeden Nachbarknoten der jeweil
ige Wert bestimmt.
    # Anhand dieses Werts wird die Liste absteigend sortiert,
    # dass der Nachbarknoten mit dem höchsten Wert an erster Stelle steht.
    # Somit wird die Tiefensuche beschleunigt.
    sortierte_liste_nachfolger = sorted(graph[aktueller_knoten],
```

```
                                        key=lambda item: self.heuristik_dfs(*item[
0][0], aktuelle_ladung_batterien, graph), reverse=True)

    # sortierte_liste_nachfolger = sorted(graph[aktueller_knoten],
    #                                     key=lambda item: len(self.erreichbareBat
terien(*item[0][0], item[0][1], restliche_batterien)), reverse=True)

    # für jeden Kindernknoten wird die Schleife aufgerufen
    for nachfolger_item in sortierte_liste_nachfolger:
        knoten, gewichtung = nachfolger_item[0]

        # falls der Knoten noch nicht besucht wurde
        if knoten in restliche_batterien:

            # Die Methode dfs wird nun rekursiv aufgerufen.

            # Dabei wird der Graph mithilfe des Moduls copy tiefenkopiert,
            # da sonst eine Veränderung des Graphen in einem Nachbarknoten, die an
deren auch beeinflussen würde.

            # Der Nachbarknoten ist dann der aktuelle_knoten.
            # Die alte Ladung des vorherigen Knoten ist die Differenz zwischen des
sen ursprüngliche Ladung und die Gewichtung zum Nachbarknoten.
            # a_alte_ladung wird mit dem Wert von alte_ladung aufgerufen, und 'rüc
kt somit eins weiter nach hinten'.

            # Das Dictionary zum Speichern der aktuellen Ladungen wird ebenfalls k
opiert,
            # da sonst dieselben Probleme beim rekursiven Aufruf auftreten würden
(wie bei graph).
            return_item = self.dfs(graph=copy.deepcopy(graph), aktueller_knoten=kn
oten,
                alte_ladung=(
                aktuelle_ladung-gewichtung),
                a_alte_ladung=alte_ladung,
                aktuelle_ladung_batterien=aktuelle_ladung_batterien.copy(),
                pfad=pfad.copy())

            # falls der rekursive Aufrufe ein Tuple zurückgibt,
            # wird dieses auch zurückgegeben
            if return_item:
                return return_item

            # 'else:' Falls nichts zurückgegeben wird, geht dieser Teilzweig ins L
eere und wird nicht weiter beachtet.
```

Main-Methode

```
def main(self):
    """ Hauptmethode des gesamten Programms
    """
    # Start der Zeitmessung
    start_zeit = time.time()

    # ein Graph der Klasse Graph wird erzeugt
    self.graph = Graph()

    # zu diesem Graphen wird für jede Ersatzbatterie alle erreichbaren, anderen Er
satzbatterien zum Graphen hinzugefügt
```

```
    for batterie in self.batterien:

        # eine Kopie der Liste wrid erstellt, von der die aktuelle Batterie entfer
nt wird
        restliche_batterien = self.batterien.copy()
        restliche_batterien.remove(batterie)

        # alle erreichbaren Batterien werden ermittelt
        erreichbare_batterien = self.erreichbareBatterien(*batterie, restliche_bat
terien)

        for erreichbare_batterie in erreichbare_batterien:
            # die erreichbaren Batterien bilden zusammen mit der aktuellen Batteri
e eine Kante,
            # wobei die aktuelle Batterie der Startknoten der Kante und die erreic
hbare Batterie der Endknoten der Kante ist

            # nur die x- und y-
Koordinaten der Batterien werden benötigt, die Ladung der Batterien nicht
            x_start, y_start = batterie[:2]
            x_ende, y_ende = erreichbare_batterie[:2]

            # verbrauchte Ladung ist die Manhattan-Distanz
            verbrauchte_ladung = self.manhattanDistanz(
                x_start, y_start, x_ende, y_ende)

            # die Gewichtung der Kante ist die verbrauchte Ladung von der aktuelle
n Batterie zur erreichbaren Batterie
            # Kante wird zum Graphen hinzugefügt
            self.graph.add_Kante((x_start, y_start),
                                 (x_ende, y_ende), verbrauchte_ladung)


    # Dasselbe wird ebenfalls für den Roboter durchgeführt:
    # Erreichbare Batterien werden ermittelt
    roboter_erreichbare_batterien = self.erreichbareBatterien(*self.roboter, self.
batterien)
    for erreichbare_batterie in roboter_erreichbare_batterien:
        x_start, y_start = self.roboter[:2]
        x_ende, y_ende = erreichbare_batterie[:2]

        verbrauchte_ladung = self.manhattanDistanz(
            x_start, y_start, x_ende, y_ende)

        # Kante wird zum Graphen hinzugefügt
        self.graph.add_Kante((x_start, y_start),
                             (x_ende, y_ende), verbrauchte_ladung)

    # Ein Dictionary zum Speichern der aktuellen Ladungen der Batterien wird erste
llt
    aktuelle_ladung_batterien = defaultdict(list)
    for batterie in self.batterien:
        x, y, ladung = batterie
        # x- und y-Koordinate stellen den Key eines Elements im Dictionary dar
        aktuelle_ladung_batterien[(x, y)] = ladung

    # auch die Startladung des Roboters wird zum Dictionary hinzugefügt
    aktuelle_ladung_batterien[self.roboter[:2]] = self.roboter[2]
```

```
    # Ein Pfad wird mit der Methode dfs ermittelt,
    # welcher die Route des Roboters angibt, damit alle Batterien am Ende leer sin
d

    # Der Pfad beinhaltet die Punkte auf dem Spielbrett, zu welchem der Roboter ge
ht.
    # Am Ende hat der Roboter noch eine restliche Ladung, welche die Methode dfs e
benfalls zurückgibt.
    # die restliche Ladung wird später dann noch "verbraucht".
    return_item = self.dfs(
        graph=self.graph,
        aktueller_knoten=self.roboter[:2],
        alte_ladung=0,
        a_alte_ladung=0,
        aktuelle_ladung_batterien=aktuelle_ladung_batterien)

    # falls kein Pfad gefunden wurde:
    if not return_item:

        # Hier wird auch die Laufzeit ermittelt
        ende_zeit = time.time()

        print(">> Die eingegebene Spielsituation ist nicht lösbar! \nKein Pfad wur
de gefunden!")

        print(f"\n>> Laufzeit des Programms: {ende_zeit-
start_zeit} Sekunden \n (Start der Zeitmessung bei Aufruf der main-Methode)")

        messagebox.showerror("Fehler", "Kein Pfad gefunden!")
        return

    pfad, restliche_ladung = return_item

    print(">> Pfad gefunden > ", pfad)

    # die Abfolge der Schritte des Roboters werden aus dem Pfad ermittelt
    # mögliche Schritte sind: nach oben, rechts, unten und links

    # in abfolge_schritte werden die Schritte für den gesamten Pfad gespeichert
    abfolge_schritte = []
    for i in range(len(pfad)):
        if i > 0:

            # zwei aufeinanderfolgende Punkte spiegeln einen Teilweg des Pfades wi
eder
            p1 = pfad[i-1]
            p2 = pfad[i]

            # die Methode findeWeg gibt eine mögliche Abfolge der Schritte für die
sen Teilweg zurück
            # die möglichen Schritte werden als Zahl dargestellt. (Näheres ist der
 Methode findeWeg oder der Dokumentation zu entnehmen)
            schritte = self.findeWeg(*p1, *p2)

            # die Schritte für den Teilweg werden zu abfolge_schritte hinzugefügt
            abfolge_schritte.extend(schritte)

    # als Liste der restlichen Batterien werden nur die x- und y-
Koordinaten aller Batterien benötigt
```

```
    batterien_x_y = list(
        map(lambda batterie: batterie[:2], self.batterien))

    # zu der Abfolge der Schritten kommen noch diejenigen Schritte hinzu,
    # die benötigt werden, um die restliche Ladung zu entladen, die der Roboter zu
m Schluss noch besitzt
    if restliche_ladung > 0:
        letzte_position = pfad[-1]

        # eine Liste mit Nachbarpunkten von der letzten Position ausgehend wird er
stellt
        liste_nachbarn1 = self.findeFreieNachbarpunkte(*letzte_position, batterien
_x_y)

        # Prüfen, ob ein Tuple in der Liste ist
        if any(isinstance(item, tuple) for item in liste_nachbarn1):

            # Liste wird so sortiert, dass eine Zahl am Anfang ist, welche ausgewä
hlt wird
            liste_nachbarn1.sort(key=lambda x: (x is None, x))
            # dies ist der erste Nachbar
            erster_nachbar = liste_nachbarn1[0]

            # eine Liste mit freien Nachbarpunkten vom ersten Nachbarn ausgehend w
ird erstellt
            liste_nachbarn2 = self.findeFreieNachbarpunkte(*erster_nachbar, batter
ien_x_y)

            # Prüfen, ob eine Tuple in der Liste ist
            if any(isinstance(item, tuple) for item in liste_nachbarn2):

                # Liste wird so sortiert, dass eine Zahl am Anfang ist, welche aus
gewählt wird
                liste_nachbarn2.sort(key=lambda x: (x is None, x))
                # dies ist der erste Nachbar
                zweiter_nachbar = liste_nachbarn2[0]

                # Schritt von letzter Position zum ersten Nachbarn
                von_letzter_pos_zu_nachbar1 = self.findeWeg(*letzte_position, *ers
ter_nachbar)[0]

                # Schritt vom ersten Nachbarn zum zweiten Nachbarn
                von_nachbar1_zu_nachbar2 = self.findeWeg(*erster_nachbar, *zweiter
_nachbar)[0]

                # Schritt vom zweiten Nachbarn zurück zum ersten
                von_nachbar2_zu_nachbar1 = self.findeWeg(*zweiter_nachbar, *erster
_nachbar)[0]

                # es wird von der letzten Position zum ersten Nachbarn gegangen
                abfolge_schritte.append(von_letzter_pos_zu_nachbar1)
                restliche_ladung -= 1
                aktuelle_position = erster_nachbar

                # Solange der Roboter noch Ladung besitzt,
                # geht dieser immer vom ersten Nachbarn zum zweiten und wieder zur
ück
                # Er geht sozusagen immer hin und her
                while restliche_ladung > 0:
```

```
                    # wenn sich der Roboter gerade auf dem Feld des ersten Nachbar
n befindet
                    # so geht er vom ersten zum zweiten Nachbarn
                    if aktuelle_position == erster_nachbar:
                        abfolge_schritte.append(von_nachbar1_zu_nachbar2)
                        aktuelle_position = zweiter_nachbar

                    # wenn sich der Roboter gerade auf dem Feld des zweiten Nachba
rn befindet
                    # so geht er vom zweiten wieder zum ersten Nachbarn
                    else:
                        abfolge_schritte.append(von_nachbar2_zu_nachbar1)
                        aktuelle_position = erster_nachbar

                    restliche_ladung -= 1

    if restliche_ladung == 1:

        # Wenn am oberen Rand (also y-Koordinate gleich 1)
        # Schritt nach unten
        if letzte_position[1] == 1:
            abfolge_schritte.append(1)

        # sonst Schritt nach oben
        else:
            abfolge_schritte.append(0)

    # Abfolge der Schritte wird von Zahlen zu deutschen Wörtern 'konvertiert'
    abfolge_schritte_deutsch = []
    for schritt in abfolge_schritte:
        if schritt == 0:
            abfolge_schritte_deutsch.append('oben')
        elif schritt == 1:
            abfolge_schritte_deutsch.append('unten')
        elif schritt == 2:
            abfolge_schritte_deutsch.append('links')
        elif schritt == 3:
            abfolge_schritte_deutsch.append('rechts')

    ende_zeit = time.time()

    print(f"\n>> Abfolge von {len(abfolge_schritte_deutsch)} Schritten für den Rob
oter in deutscher Sprache: \n > {abfolge_schritte_deutsch}")

    print(f"\n>> Laufzeit des Programms: {ende_zeit-
start_zeit} Sekunden \n (Start der Zeitmessung bei Aufruf der main-Methode)")
```

(Es folgt die Initiierung eines Objekts der Klasse ‚Environment‘, sowie der Aufruf der Methode ‚step(…)‘ für jedes Element in der Liste ‚abfolge_schritte‘)

Stromrallye spielen

Lösungsidee

Ziel dieser Aufgabe ist es, eine eigene Spielsituation zu erzeugen, die nach gegebenen Regeln gelöst werden kann. Für einen menschlichen Spieler sollen diese aber schwer zu lösen sein.

Die Idee hinter dieser Aufgabe ist, einen beliebigen Pfad zu erstellen, den der Roboter für die Lösung fahren soll. Dabei soll das Zielfeld dieses simulierten Pfades die Startposition dieses simulierten Pfades sein.

Zu Beginn wird anhand des Schwierigkeitsgrades die Anzahl *n* der vorkommenden Batterien (Ersatzbatterien + Bordbatterie des Roboters) und der Bereich der Ladungen festgelegt:

- leicht: 5 Batterien, mind. Ladungen 3
- mittel: 10 Batterien, mind. Ladungen 5
- schwer: 15 Batterien, mind. Ladungen 10

Der Benutzer kann bei der Größe des Spielfelds zwischen 10x10, 12x12 und 14x14 Feldern wählen.

Das Startfeld des simulierenden Pfades wird zufällig ausgewählt.

Folgende Anweisungen werden ausgeführt, bis alle Batterien (Anzahl=*n*) gelegt sind:

- Bestimme zufällig die Anzahl der Ladung *s* aus dem vorher bestimmten Bereich der Mindestladungen (z.B. mittel → 5 bis 10 Ladungen).
- Wähle von der aktuellen Position ausgehend zufällig ein freies Nachbarfeld *f* aus, das als nächstes besucht werden soll.
  Das freie Nachbarfeld *f* ist die neue aktuelle Position.
  Speichere alle besuchten Felder *f* in der Liste *q*
  → Wiederhole diese Bestimmung eines neuen Nachbarfelds *f* und somit die Auswahl einer neuen Position *s*-mal.
- Auf dem aktuellen Feld nach dieser *s*-maligen Wiederholung soll nun eine Batterie mit der Ladung *s* gelegt werden.
  Überprüfe dafür, ob das aktuelle Feld bereits im simulierten Pfad besucht wurde und somit in *q* oder q_{gesamt} ist.
  Ist dies der Fall, so wiederhole die obige Schleife erneut unter Erhöhung von *s* um 1 und wähle eine neue Position aus.

  Wenn nicht, so lege auf das aktuelle Feld eine neue Batterie.
  Speichere die besuchten Nachbarfelder *q* in der gesamten Liste q_{gesamt}.

Wurden diese Anweisungen nun *n*-mal durchgeführt, so sind alle Batterien auf dem Spielfeld verteilt worden. Die letzte gelegte Batterie *b* auf dem Feld *p* ist die Bordbatterie des Roboters auf dem Startfeld *p* mit Ladung der Batterie *b*.

Die erstellte Spielumgebung wird nun im Textformat und im Format des BwInf ausgegeben und ebenfalls graphisch dargestellt.

Erweiterung:

Die Umkehrung der Liste q_{gesamt} speichert nun den Weg, den der Roboter als mögliche Lösung der erstellen Spielsituation verwenden kann, um alle Batterien leer zu fahren.

Diese kann, wenn der Benutzer es wünscht, als deutschsprachige Schrittanweisungen ausgegeben werden. Auch eine Visualisierung der Schritte des Roboters in der Umgebung kann erfolgen.

Umsetzung

Die Lösungsidee wird objektorientiert in Python implementiert.
Die für die Erstellung der Spielumgebung relevante Klasse ist die Klasse *SpielErzeugen*. Eine Visualisierung erfolgt durch die Klasse *Environment*.
Das Eingabefenster für die Eingabe des Schwierigkeitsgrades und der Größe des Spielfelds ist durch die Klasse *EingabeFenster* gegeben.

Methode zur Bestimmung eines zufälligen Nachbarfelds

Die Methode *randomStep(x_akt, y_akt, vorhandene_batterien)* wählt zufällig ein freies Nachbarfeld aus, das besucht werden kann.
Wichtig dabei ist, dass das zu besuchende Nachbarfeld nicht durch eine Ersatzbatterie bereits belegt sein sollte.

Die Liste *vorhandene_batterien* speichert die x- und y-Koordinaten der bereits festgelegten Positionen der Ersatzbatterien.

Die möglichen Nachbarfelder werden in der Liste *schritte* gespeichert.

Für die benachbarten Felder *unten*, *links*, *rechts* und *oben* werden als erstes die Koordinaten ermittelt. Wenn sich die benachbarten Felder in *vorhandene_batterien* befinden, so wird den jeweiligen Variablen *None* zugewiesen.

Da Felder an den Rändern bzw. in den Ecken des Spielfelds nicht alle Nachbarfelder haben können, soll die Überprüfung durch eine Reihe von if-Bedingungen erfolgen, welche die vorliegende Position des Felds eingrenzen.

- y_akt ist größer als 1:
  → aktuelle Position ist nicht am oberen Rand
  Zur Liste *schritte* wird das Feld *oben* hinzugefügt.
- *y_akt* ist kleiner als *self.size*:
  → aktuelle Position ist nicht unten am Rand
  Zur Liste *schritte* wird das Feld *unten* hinzugefügt.
- *x_akt* ist größer als 1:
  → aktuelle Position ist nicht am linken Rand
  Zur Liste *schritte* wird das Feld *links* hinzugefügt.
- *x_akt* ist kleiner als *self.size*:
  → aktuelle Position ist nicht rechts am Rand
  Zur Liste *schritte* wird das Feld *rechts* hinzugefügt.

Anschließend werden alle Elemente mit dem Wert *None* von *schritte* entfernt.

Falls noch mögliche Nachbarfelder in *schritte* enthalten sind, wird ein zufälliger Index ausgewählt und über diesen Index ein Nachbarfeld von *schritte* ausgewählt und zurückgegeben.

Falls *schritte* jedoch leer ist, wird *None* zurückgegeben.

Initialisierungsmethode

Die Methode *__init__(schwierigkeit, größe_spielfeld)* wird zur Initialisierung eines Objekts der Klasse *SpielErzeugen* aufgerufen und kann dabei (vom Eingabefenster) folgende Werte für die beiden Parameter erhalten:
Bedeutung des Werts für den Parameter *schwierigkeit*:

- 1: leichte Spielsituation
- 2: mittlere Spielsituation
- 3: schwere Spielsituation

Bedeutung des Werts für den Parameter *größe_spielfeld*:

- 10: 10x10 Spielfeld
- 12: 12x12 Spielfeld
- 14: 14x14 Spielfeld

Die jeweiligen Werte werden in den Klassenvariablen *self.schwierigkeit* (Schwierigkeitsgrad) und *self.size* (Größe des Spielfelds) gespeichert.

Anschließend werden die x- und y-Koordinate des Startpunkts mithilfe des Moduls *random* im Bereich von 1 bis *self.size* zufällig ausgewählt. Dieser Startpunkt ist ebenfalls die aktuelle Position P_{akt}

Danach wird in der Klassenvariable *self.bereich_schritte* der Bereich der Schritte (und somit die Anzahl der Ladungen der Batterien) anhand des Werts von *self.schwierigkeit* festgelegt:

- 1 → Bereich von 1 bis 5
- 2 → Bereich von 5 bis 10
- 3 → Bereich von 10 bis 15

Ebenso wird die Anzahl der Batterien mithilfe des Schwierigkeitsgrades *self.anzahl_batterien* festgelegt:

- 1 → 5 Batterien
- 2 → 10 Batterien
- 3 → 15 Batterien

Alle Batterien werden in der Liste *self.batterien* und die gesamten besuchten Felder in der Liste *gesamte_positionen* gespeichert.

Mithilfe einer for-Schleife werden nun folgende Anweisungen *self.anzahl_batterien-mal* wiederholt:

- Speichere die x- und y-Koordinaten der bereits erstellten Batterien *self.batterien* in der Liste *batterien_x_y*
- Wähle mithilfe des Moduls *random* eine zufällige Anzahl der Schritte *s* im vorher bestimmten Bereich *self.bereich_schritte* aus.
- Erstelle eine leere Liste *positionen*, in der alle besuchten Felder gespeichert werden, bevor eine neue Batterie gelegt wird.
- Wiederhole mithilfe eines Zählers, der den Wert *s* besitzt, folgende while-Schleife:
    - *„der letzte Check vor dem Batterie-Legen“*: Ist der Zähler aktuell auf 0, so wurden alle Schritte durchgeführt und die Batterie wird im nächsten Schritt auf die aktuelle Position P_{akt} gelegt.
      Nun wird überprüft, ob P_{akt} bereits durch Schritte besucht wurde und sich somit entweder in der Liste *gesamte_positionen* oder in *positionen* befindet.
      Ist dies der Fall, so wird der Zähler um eins und die Anzahl der Schritte *s* um eins erhöht und das letzte Element von *positionen* entfernt, da sich die Batterie auf einem besuchten Feld nicht befinden soll.

      Wenn nicht, so wird der Zähler um eins verringert.
      Am Ende dieser if-Bedingung wird mit ‚*continue*‘ die nächste Iteration fortgesetzt.

- Bestimme mithilfe der Methode *randomStep* von der aktuellen Position P_{akt} ausgehend ein neues Nachbarfeld.
  Existiert kein Nachbarfeld und es wird *None* zurückgegeben, so wird das letzte Element von *positionen* entfernt und der Zähler der while-Schleife um eins erhöht.

  Existiert ein Nachbarfeld, so wird dieses Feld zu *positionen* hinzugefügt und der Zähler um eins verringert.
- Aktualisiere nun die aktuelle Position P_{akt} indem das zuletzt hinzugefügte Feld der Liste *positionen* abgefragt wird.
  Ist die Liste *positionen* jedoch leer, so wird P_{akt} das zuletzt hinzugefügte Feld der Liste *gesamte_positionen* zugewiesen.

- Ist die while-Schleife beendet, so wird zu der Liste der gesamten Batterien *self.batterien* eine neue Batterie mit der aktuellen Position P_{akt} als Feld und mit *s* als Ladung hinzugefügt.

Ist die for-Schleife beendet, so wird die Liste *gesamte_positionen* umgekehrt und dem Klassenattribut *self.gesamte_positionen* zugeordnet. *Self.gesamte_positionen* speichert nun den Weg, den der Roboter als mögliche Lösung der Spielsituation verwenden kann, um alle Batterien leer zu fahren.

Nun wird die Methode *erzeugeUmgebung()* aufgerufen. In dieser Methode wird die Zeitmessung, die am Anfang der Methode *__init__(...)* begonnen hat, beendet und die benötigte Laufzeit ausgegeben.
Ebenfalls wird nun eine Konsolenausgabe mit den „Daten“ der Spielsituation erzeugt, wobei die zuletzt hinzugefügte Batterie der Liste *self.batterien* den Roboter darstellt.
Die „Daten“ der Spielumgebung werden ebenfalls im korrekten Format des BwInf ausgegeben.
Ferner wird ein Objekt der Klasse *Environment* initiiert, welches die Daten der Spielumgebung erhält und diese somit visualisieren kann.

Methoden zum Zeigen der Lösung der Spielsituation

Fordert der Benutzer das Programm durch Betätigen eines weiteren Buttons des Eingabefenster dazu auf, die Lösung für die eben erstellte Spielsituation zu zeigen, wird die Methode *zeigeLösung()* aufgerufen.

Durch die Methode *wegInAnweisungen(punkte)* werden alle gegebenen Punkte in eine Abfolge von Schrittanweisungen für den Roboter konvertiert. Diese Abfolge von Schritten sollen in der Liste *abfolge_schritte* gespeichert werden.

Für jedem Punkt *P* in *punkte* (außer dem ersten Punkt) wird ein Teil-Weg konstruiert, welcher genau einen Schritt des Roboters beschreibt:

- Von dem Punkt *P* und seinem Vorgänger *P‘* werden die Differenzen der x- und y-Koordinaten berechnet: $\Delta x = x_{P'} - x_P$ und $\Delta y = y_{P'} - y_P$
- Je nach Differenz der Koordinaten muss eine bestimmte Bewegung ausgeführt werden, um von *P‘* zu *P* zu gelangen:
  - $\Delta x > 0: bewegung = 3$
  - $\Delta x < 0: bewegung = 2$
  - $\Delta y > 0: bewegung = 1$
  - $\Delta y < 0: bewegung = 0$
- Die ermittelte Variable *bewegung* wird zur Liste *abfolge_schritte* hinzugefügt.

Am Ende wird die Liste *abfolge_schritte* zurückgegeben.

Nachdem die Methode *wegInAnweisungen(...)* die Liste *self.gesamte_positionen* in eine Abfolge von Schrittanweisungen *schrittanweisungen_roboter_lösung* konvertiert hat, wird diese Abfolge noch von Zahlen zu deutschen Wörtern ‚übersetzt‘:

- 0 → ‚oben‘
- 1 → ‚unten‘
- 2 → ‚links‘
- 3 → ‚rechts‘

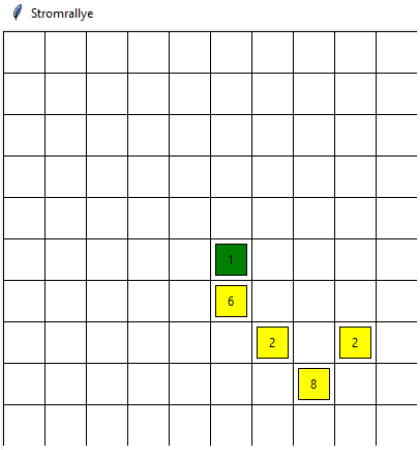
Die Abfolge der deutschen Wörter und die Liste *self.gesamte_positionen* werden in der Konsole ausgegeben.

Mithilfe der Liste *schrittanweisungen_roboter_lösung* werden die Schritte des Roboters durch Aufruf der Methode *step(schritt)* des Objekts der Klasse *Environment* im GUI visualisiert. So kann der Spieler die Lösung nachverfolgen und verstehen.

Beispiele

Schwierigkeitsgrad leicht mit 10x10 Spielfeld:

>> Laufzeit des Programms: 0.0 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)



Screenshot der erzeugten Spielumgebung

>> Spielgröße: 10
>> Startposition des Roboters (6, 6, 1)
>> Anzahl der Batterien: 4
>> Koordinaten der Batterien: [(9, 8, 2), (8, 9, 8), (7, 8, 2), (6, 7, 6)]

>> Ausgabe im BwInf-Format:
10
6,6,1
4
9,8,2
8,9,8
7,8,2
6,7,6

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(6, 6), (6, 7), (7, 7), (8, 7), (7, 7), (8, 7), (7, 7), (7, 8), (8, 8), (8, 9), (9, 9), (9, 10), (9, 9), (9, 10), (10, 10), (10, 9), (9, 9), (9, 8), (9, 7), [9, 8]]

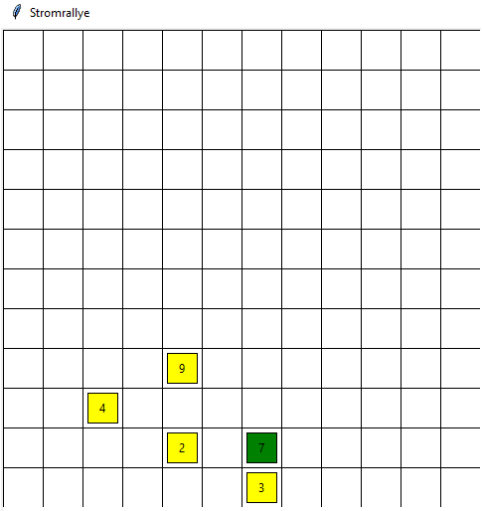
>> Abfolge von 19 Schritten für den Roboter in deutscher Sprache:
> ['unten', 'rechts', 'rechts', 'links', 'rechts', 'links', 'unten', 'rechts', 'unten', 'rechts', 'unten', 'oben', 'unten', 'rechts', 'oben', 'links', 'oben', 'oben', 'unten']

Schwierigkeitsgrad leicht mit 12x12 Spielfeld:

>> Laufzeit des Programms: 0.0 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 12
>> Startposition des Roboters (7, 11, 7)
>> Anzahl der Batterien: 4
>> Koordinaten der Batterien: [(3, 10, 4), (5, 9, 9), (5, 11, 2), (7, 12, 3)]

>> Ausgabe im BwInf-Format:
12
7,11,7
4
3,10,4
5,9,9
5,11,2
7,12,3



Screenshot der erzeugten Spielumgebung

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(7, 11), (6, 11), (6, 12), (5, 12), (6, 12), (5, 12), (6, 12), (7, 12), (6, 12), (6, 11), (5, 11), (5, 10), (5, 9), (4, 9), (4, 8), (4, 9), (4, 10), (4, 9), (4, 8), (4, 9), (3, 9), (3, 10), (4, 10), (5, 10), (6, 10), [7, 10]]

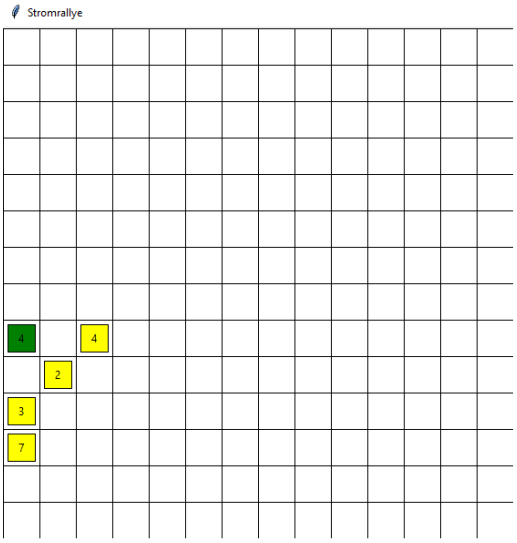
>> Abfolge von 25 Schritten für den Roboter in deutscher Sprache:
> ['links', 'unten', 'links', 'rechts', 'links', 'rechts', 'rechts', 'links', 'oben', 'links', 'oben', 'oben', 'links', 'oben', 'unten', 'unten', 'oben', 'oben', 'unten', 'links', 'unten', 'rechts', 'rechts', 'rechts', 'rechts']

Schwierigkeitsgrad leicht mit 14x14 Spielfeld:

>> Laufzeit des Programms: 0.0010013580322265625 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 14
>> Startposition des Roboters (1, 9, 4)
>> Anzahl der Batterien: 4
>> Koordinaten der Batterien: [(1, 12, 7), (1, 11, 3), (2, 10, 2), (3, 9, 4)]

>> Ausgabe im BwInf-Format:
14
1,9,4
4
1,12,7
1,11,3
2,10,2
3,9,4



Screenshot der erzeugten Spielumgebung

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(1, 9), (2, 9), (2, 8), (2, 9), (3, 9), (2, 9), (2, 8), (2, 9), (2, 10), (2, 11), (1, 11), (2, 11), (2, 12), (1, 12), (1, 13), (1, 14), (2, 14), (1, 14), (1, 13), (1, 14), [1, 13]]

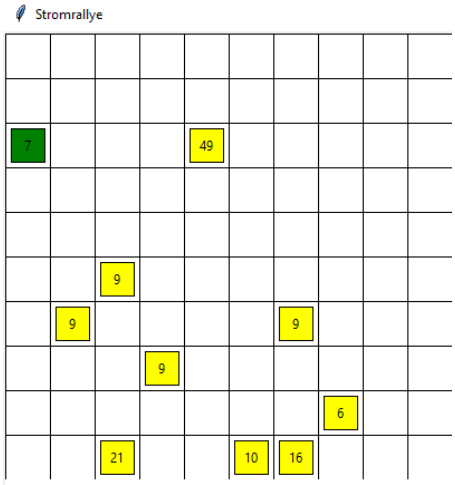
>> Abfolge von 20 Schritten für den Roboter in deutscher Sprache:
> ['rechts', 'oben', 'unten', 'rechts', 'links', 'oben', 'unten', 'unten', 'unten', 'links', 'rechts', 'unten', 'links', 'unten', 'unten', 'rechts', 'links', 'oben', 'unten', 'oben']

Schwierigkeitsgrad mittel mit 10x10 Spielfeld:

>> Laufzeit des Programms: 0.0010018348693847656 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 10
>> Startposition des Roboters (1, 3, 7)
>> Anzahl der Batterien: 9
>> Koordinaten der Batterien: [(7, 10, 16), (8, 9, 6), (7, 7, 9), (3, 6, 9), (4, 8, 9), (6, 10, 10), (3, 10, 21), (5, 3, 49), (2, 7, 9)]

>> Ausgabe im BwInf-Format:
10
1,3,7
9
7,10,16
8,9,6
7,7,9
3,6,9
4,8,9
6,10,10
3,10,21
5,3,49
2,7,9



Screenshot der erzeugten Spielumgebung

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(1, 3), (1, 4), (2, 4), (2, 5), (1, 5), (2, 5), (2, 6), (2, 7), (1, 7), (1, 6), (1, 5), (2, 5), (2, 4), (3, 4), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (6, 7), (5, 7), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (4, 10), (4, 9), (4, 10), (4, 9), (4, 10), (4, 9), (4, 10), (4, 9), (4, 10), (5, 10), (5, 9), (4, 9), (4, 10), (4, 9), (4, 10), (5, 10), (4, 10), (4, 9), (3, 9), (4, 9), (5, 9), (5, 8), (5, 7), (6, 7), (6, 6), (5, 6), (5, 7), (6, 7), (6, 8), (7, 8), (6, 8), (6, 7), (5, 7), (5, 8), (5, 9), (4, 9), (3, 9), (3, 10), (4, 10), (4, 9), (4, 10), (5, 10), (5, 9), (5, 10), (5, 9), (5, 10), (5, 9), (4, 9), (3, 9), (4, 9), (5, 9), (5, 10), (4, 10), (5, 10), (5, 9), (5, 10), (4, 10), (5, 10), (6, 10), (5, 10), (5, 9), (6, 9), (7, 9), (7, 8), (6, 8), (5, 8), (5, 9), (5, 8), (4, 8), (4, 9), (3, 9), (3, 8), (3, 7), (3, 8), (2, 8), (3, 8), (3, 7), (3, 6), (4, 6), (5, 6), (5, 5), (5, 4), (5, 5), (6, 5), (6, 6), (6, 7), (7, 7), (8, 7), (9, 7), (9, 6), (9, 7), (8, 7), (8, 8), (8, 7), (8, 8), (8, 9), (9, 9), (10, 9), (9, 9), (9, 10), (8, 10), (7, 10), (8, 10), (9, 10), (8, 10), (9, 10), (10, 10), (9, 10), (8, 10), (9, 10), (10, 10), (9, 10), (10, 10), (10, 9), (9, 9), (9, 10), (9, 9), [8, 9]]

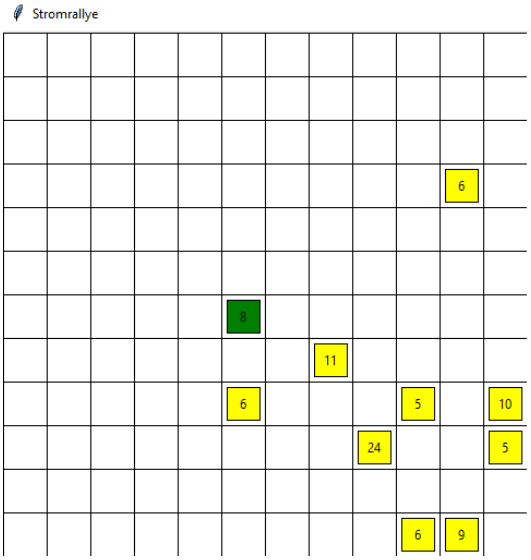
>> Abfolge von 145 Schritten für den Roboter in deutscher Sprache:
> ['unten', 'rechts', 'unten', 'links', 'rechts', 'unten', 'unten', 'links', 'oben', 'oben', 'rechts', 'oben', 'rechts', 'oben', 'rechts', 'rechts', 'unten', 'unten', 'unten', 'unten', 'rechts', 'links', 'oben', 'unten', 'unten', 'unten', 'unten', 'links', 'oben', 'unten', 'oben', 'unten', 'oben', 'unten', 'oben', 'unten', 'rechts', 'oben', 'links', 'unten', 'oben', 'unten', 'rechts', 'links', 'oben', 'links', 'rechts', 'rechts', 'oben', 'oben', 'rechts', 'oben', 'links', 'unten', 'rechts', 'unten', 'rechts', 'links', 'oben', 'links', 'unten', 'unten', 'links', 'links', 'unten', 'rechts', 'oben', 'unten', 'rechts', 'oben', 'unten', 'oben', 'unten', 'oben', 'links', 'links', 'rechts', 'rechts', 'unten', 'links', 'rechts', 'oben', 'unten', 'links', 'rechts', 'rechts', 'links', 'oben', 'rechts', 'rechts', 'oben', 'links', 'links', 'unten', 'oben', 'links', 'unten', 'links', 'oben', 'oben', 'unten', 'links', 'rechts', 'oben', 'oben', 'rechts', 'rechts', 'oben', 'oben', 'unten', 'rechts', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'oben', 'unten', 'links', 'unten', 'oben', 'unten', 'unten', 'rechts', 'rechts', 'links', 'unten', 'links', 'links', 'rechts', 'rechts', 'links', 'rechts', 'rechts', 'links', 'links', 'rechts', 'rechts', 'links', 'rechts', 'oben', 'links', 'unten', 'oben', 'links']

Schwierigkeitsgrad mittel mit 12x12 Spielfeld:

>> Laufzeit des Programms: 0.0 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 12
>> Startposition des Roboters (6, 7, 8)
>> Anzahl der Batterien: 9
>> Koordinaten der Batterien: [(11, 4, 6), (8, 8, 11), (10, 12, 6), (10, 9, 5), (12, 10, 5), (11, 12, 9), (12, 9, 10), (9, 10, 24), (6, 9, 6)]

>> Ausgabe im BwInf-Format:
12
6,7,8
9
11,4,6
8,8,11
10,12,6
10,9,5
12,10,5
11,12,9
12,9,10
9,10,24
6,9,6



Screenshot der erzeugten Spielumgebung

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(6, 7), (6, 8), (7, 8), (7, 9), (7, 8), (7, 9), (7, 8), (6, 8), (6, 9), (6, 10), (7, 10), (8, 10), (8, 9), (9, 9), (9, 10), (9, 11), (10, 11), (11, 11), (12, 11), (11, 11), (11, 10), (10, 10), (10, 11), (9, 11), (10, 11), (11, 11), (12, 11), (12, 12), (12, 11), (11, 11), (10, 11), (10, 10), (10, 11), (9, 11), (10, 11), (11, 11), (11, 10), (11, 9), (12, 9), (11, 9), (11, 10), (11, 9), (11, 8), (11, 9), (11, 10), (11, 11), (12, 11), (12, 12), (11, 12), (11, 11), (12, 11), (11, 11), (12, 11), (11, 11), (11, 10), (11, 11), (11, 10), (12, 10), (12, 11), (11, 11), (11, 10), (10, 10), (10, 9), (10, 10), (10, 11), (9, 11), (10, 11), (10, 12), (10, 11), (9, 11), (8, 11), (8, 10), (8, 9), (8, 8), (8, 7), (9, 7), (8, 7), (7, 7), (8, 7), (8, 6), (9, 6), (10, 6), (11, 6), (11, 5), (11, 4), (11, 3), (12, 3), (12, 2), (12, 3), (12, 2), [12, 3]]

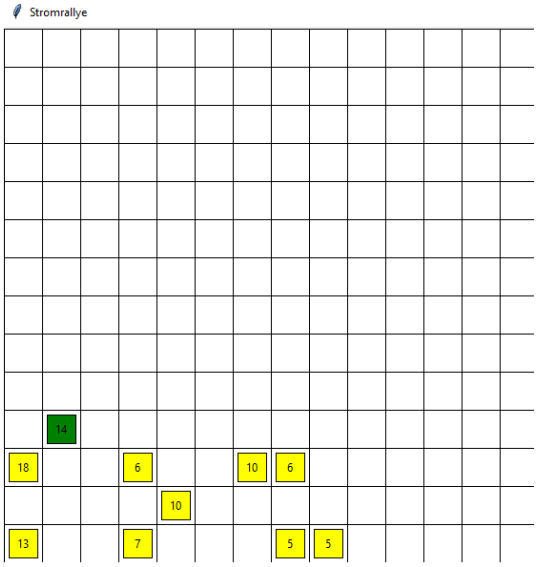
>> Abfolge von 90 Schritten für den Roboter in deutscher Sprache:
> ['unten', 'rechts', 'unten', 'oben', 'unten', 'oben', 'links', 'unten', 'unten', 'rechts', 'rechts', 'oben', 'rechts', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'links', 'oben', 'links', 'unten', 'links', 'rechts', 'rechts', 'rechts', 'unten', 'oben', 'links', 'links', 'oben', 'unten', 'links', 'rechts', 'rechts', 'oben', 'oben', 'rechts', 'links', 'unten', 'oben', 'oben', 'unten', 'unten', 'unten', 'rechts', 'unten', 'links', 'oben', 'rechts', 'links', 'rechts', 'links', 'oben', 'unten', 'oben', 'rechts', 'unten', 'links', 'oben', 'links', 'oben', 'unten', 'unten', 'links', 'rechts', 'unten', 'oben', 'links', 'links', 'oben', 'oben', 'oben', 'oben', 'rechts', 'links', 'links', 'rechts', 'oben', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'rechts', 'oben', 'unten', 'oben', 'unten']

Schwierigkeitsgrad mittel mit 14x14 Spielfeld:

>> Laufzeit des Programms: 0.0009982585906982422 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 14
>> Startposition des Roboters (2, 11, 14)
>> Anzahl der Batterien: 9
>> Koordinaten der Batterien: [(8, 14, 5), (5, 13, 10), (8, 12, 6), (9, 14, 5), (7, 12, 10), (4, 14, 7), (4, 12, 6), (1, 14, 13), (1, 12, 18)]

>> Ausgabe im BwInf-Format:
14
2,11,14
9
8,14,5
5,13,10
8,12,6
9,14,5
7,12,10
4,14,7
4,12,6
1,14,13
1,12,18



Screenshot der erzeugten Spielstituation

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(2, 11), (2, 12), (2, 13), (3, 13), (3, 14), (2, 14), (2, 13), (2, 14), (3, 14), (2, 14), (3, 14), (2, 14), (2, 13), (1, 13), (1, 12), (2, 12), (2, 13), (2, 14), (3, 14), (2, 14), (2, 13), (1, 13), (2, 13), (2, 14), (2, 13), (2, 12), (2, 13), (2, 14), (2, 13), (3, 13), (2, 13), (1, 13), (1, 14), (2, 14), (2, 13), (3, 13), (2, 13), (3, 13), (4, 13), (3, 13), (4, 13), (3, 13), (3, 14), (3, 13), (3, 12), (4, 12), (3, 12), (2, 12), (2, 13), (2, 14), (3, 14), (4, 14), (5, 14), (6, 14), (6, 13), (6, 12), (6, 11), (7, 11), (7, 12), (7, 13), (8, 13), (9, 13), (8, 13), (9, 13), (8, 13), (9, 13), (9, 12), (9, 13), (9, 14), (9, 13), (9, 12), (9, 13), (8, 13), (8, 12), (8, 13), (7, 13), (6, 13), (6, 12), (5, 12), (5, 13), (6, 13), (6, 14), (6, 13), (6, 12), (6, 13), (7, 13), (7, 14), (7, 13), (8, 13), (8, 14), (7, 14), (6, 14), (6, 13), (7, 13), [7, 14]]

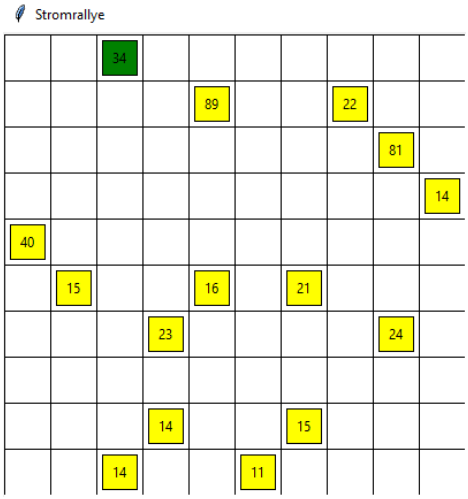
>> Abfolge von 94 Schritten für den Roboter in deutscher Sprache:
> ['unten', 'unten', 'rechts', 'unten', 'links', 'oben', 'unten', 'rechts', 'links', 'rechts', 'links', 'oben', 'links', 'oben', 'rechts', 'unten', 'unten', 'rechts', 'links', 'oben', 'links', 'rechts', 'unten', 'oben', 'oben', 'unten', 'unten', 'oben', 'rechts', 'links', 'links', 'unten', 'rechts', 'oben', 'rechts', 'links', 'rechts', 'rechts', 'links', 'rechts', 'links', 'unten', 'oben', 'oben', 'rechts', 'links', 'links', 'unten', 'unten', 'rechts', 'rechts', 'rechts', 'rechts', 'oben', 'oben', 'oben', 'rechts', 'unten', 'unten', 'rechts', 'rechts', 'links', 'rechts', 'links', 'rechts', 'oben', 'unten', 'unten', 'oben', 'oben', 'unten', 'links', 'oben', 'unten', 'links', 'links', 'oben', 'links', 'unten', 'rechts', 'unten', 'oben', 'oben', 'unten', 'rechts', 'unten', 'oben', 'rechts', 'unten', 'links', 'links', 'oben', 'rechts', 'unten']

Schwierigkeitsgrad schwer mit 10x10 Spielfeld:

>> Laufzeit des Programms: 0.003998994827270508 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 10
>> Startposition des Roboters (3, 1, 34)
>> Anzahl der Batterien: 14
>> Koordinaten der Batterien: [(3, 10, 14), (4, 9, 14), (6, 10, 11), (4, 7, 23), (5, 6, 16), (7, 9, 15), (9, 7, 24), (7, 6, 21), (9, 3, 81), (8, 2, 22), (10, 4, 14), (5, 2, 89), (2, 6, 15), (1, 5, 40)]

>> Ausgabe im BwInf-Format:
10
3,1,34
14
3,10,14
4,9,14
6,10,11
4,7,23
5,6,16
7,9,15
9,7,24
7,6,21
9,3,81
8,2,22
10,4,14
5,2,89
2,6,15
1,5,40



Screenshot der erzeugten Spielsituation

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (2, 8), (2, 9), (2, 10), (2, 9), (2, 10), (2, 9), (2, 8), (2, 7), (3, 7), (3, 6), (3, 7), (3, 6), (3, 5), (3, 4), (3, 3), (3, 4), (3, 3), (3, 2), (2, 2), (2, 3), (2, 4), (1, 4), (2, 4), (1 , 4), (2, 4), (1, 4), (1, 5), (1, 6), (1, 7), (1, 6), (1, 7), (1, 6), (1, 7), (2, 7), (1, 7), (1, 8), (1, 9), (2, 9), (1, 9), (2, 9), (2, 8), (1, 8), (2, 8), (2, 7), (1, 7), (1, 6), (1, 7), (2, 7), (2, 8), (2, 7), (2, 8), (2, 9), (2, 10), (2, 9), (2, 10), (2, 9), (2, 8), (3, 8), (3, 7), (3, 8), (3, 7), (2, 7), (3, 7), (3, 8), (3, 7), (2, 7), (2, 6), (1, 6), (1, 7), (2, 7), (3, 7), (3, 6), (3, 5), (3, 4), (4, 4), (4, 3), (5, 3), (5, 4), (4, 4), (4, 3), (5, 3), (5, 2), (6, 2), (7, 2), (7, 1), (7, 2), (7, 1), (8, 1), (9, 1), (9, 2), (10, 2), (9, 2), (10, 2), (10, 3), (10, 2), (10, 3), (10, 2), (10, 3), (10, 2), (9, 2), (9, 1), (10, 1), (9, 1), (9, 2), (9, 1), (10, 1), (9, 1), (9, 2), (10, 2), (9, 2), (9, 1), (9, 2), (10, 2), (10, 3), (10, 2), (10, 1), (10, 2), (10, 3), (10, 2), (10, 1), (10, 2), (9, 2), (10, 2), (10, 1), (9, 1), (9, 2), (10, 2), (10, 1), (10, 2), (9, 2), (10, 2), (9, 2), (10, 2), (10, 3), (10, 2), (10, 3), (10 , 2), (10, 1), (10, 2), (10, 3), (10, 2), (10, 3), (10, 2), (10, 3), (10, 2), (10, 1), (10, 2), (10, 3), (10, 2), (10, 1), (9, 1), (9, 2), (9, 1), (8, 1), (7, 1), (8, 1), (7, 1), (7, 2), (7, 1), (6, 1), (5, 1), (6, 1), (6, 2), (6, 3), (5, 3), (6, 3), (6, 4), (7, 4), (8, 4), (9, 4), (10, 4), (10, 3), (10, 2), (9, 2), (9, 1), (10, 1), (9, 1), (10, 1), (9, 1), (8, 1), (9, 1), (10, 1), (10, 2), (9, 2), (8, 2), (9, 2), (10, 2), (10, 1), (10, 2), (10, 1), (10, 2), (10, 3), (10, 2), (10, 1), (10, 2), (9, 2), (9, 1), (8, 1), (9, 1), (10, 1), (9, 1), (9, 2), (10, 2), (10, 3), (10, 2), (10, 3), (9, 3), (9, 4), (9, 5), (8, 5), (9, 5), (8, 5), (9, 5), (10, 5), (9, 5), (8, 5), (8, 6), (8, 7), (7, 7), (8, 7), (8, 8), (7, 8), (6, 8), (7, 8), (8, 8), (7, 8), (6, 8), (6, 7), (7, 7), (6, 7), (6, 8), (6, 7), (6, 8), (7, 8), (6, 8), (7, 8), (6, 8), (6, 7), (5, 7), (6, 7), (5, 7), (5, 8), (5, 7), (6, 7), (5, 7), (5, 8), (4, 8), (5, 8), (5, 9), (5, 10), (5, 9), (5, 10), (4, 10), (5, 10), (5, 9), (6, 9), (6, 8), (6, 7), (6, 8), (7, 8), (6, 8), (5, 8), (5, 7), (5, 8), (4, 8), (5, 8), (6, 8), (7, 8), (7, 7), (8, 7), (8, 6), (9, 6), (8, 6), (8, 5), (7, 5), (8, 5), (9, 5), (10, 5), (9, 5), (8, 5), (9, 5), (9, 4), (8, 4), (7, 4), (7, 5), (7, 4), (7, 5), (7, 6), (8, 6), (8, 5), (8, 6), (9, 6), (10, 6), (9, 6), (9, 5), (9, 6), (10, 6), (10,

7), (10, 6), (9, 6), (8, 6), (8, 5), (7, 5), (8, 5), (8, 4), (8, 5), (9, 5), (9, 6), (9, 7), (8, 7), (7, 7), (7, 8), (6, 8), (6, 9), (5, 9), (6, 9), (6, 8), (7, 8), (7, 7), (7, 8), (8, 8), (7, 8), (8, 8), (7, 8), (7, 7), (8, 7), (8, 8), (8, 9), (8, 10), (9, 10), (8, 10), (8, 9), (7, 9), (7, 8), (6, 8), (6, 7), (5, 7), (5, 8), (6, 8), (6, 7), (6, 8), (6, 7), (6, 6), (6, 5), (6, 4), (5, 4), (5, 5), (5, 6), (5, 7), (6, 7), (6, 8), (5, 8), (5, 9), (6, 9), (6, 8), (6, 7), (5, 7), (6, 7), (5, 7), (5, 8), (5, 7), (6, 7), (5, 7), (4, 7), (4, 8), (5, 8), (5, 9), (6, 9), (5, 9), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (4, 10), (5, 10), (6, 10), (6, 9), (6, 8), (6, 9), (6, 8), (7, 8), (6, 8), (5, 8), (5, 9), (5, 8), (4, 8), (4, 9), (3, 9), (2, 9), (2, 10), (1, 10), (2, 10), (2, 9), (3, 9), (2, 9), (1, 9), (2, 9), (3, 9), (3, 8), (3, 9), (3, 10), (2, 10), (1, 10), (1, 9), (1, 10), (2, 10), (2, 9), (2, 8), (1, 8), (1, 9), (1, 8), (1, 9), (2, 9), (3, 9), [4, 9]]

(Die Abfolge der Schritte in deutscher Sprache wurde aufgrund von Platzgründen weggelassen.)

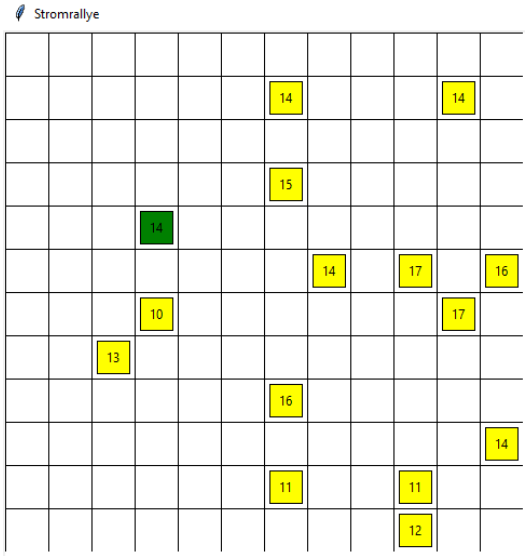
Schwierigkeitsgrad schwer mit 12x12 Spielfeld:

>> Laufzeit des Programms: 0.0019981861114501953 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 12
>> Startposition des Roboters (4, 5, 14)
>> Anzahl der Batterien: 14
>> Koordinaten der Batterien: [(11, 2, 14), (10, 6, 17), (12, 6, 16), (7, 4, 15), (7, 2, 14), (11, 7, 17), (10, 11, 11), (7, 11, 11), (10, 12, 12), (12, 10, 14), (8, 6, 14), (7, 9, 16), (3, 8, 13), (4, 7, 10)]

>> Ausgabe im BwInf-Format:

```
12
4,5,14
14
11,2,14
10,6,17
12,6,16
7,4,15
7,2,14
11,7,17
10,11,11
7,11,11
10,12,12
12,10,14
8,6,14
7,9,16
3,8,13
4,7,10
```



Screenshot der erzeugten Spielsituation

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(4, 5), (4, 6), (5, 6), (6, 6), (7, 6), (7, 7), (6, 7), (6, 8), (5, 8), (5, 9), (5, 10), (5, 9), (4, 9), (4, 8), (4, 7), (4, 8), (5, 8), (4, 8), (5, 8), (5, 9), (6, 9), (5, 9), (5, 8), (4, 8), (3, 8), (4, 8), (5, 8), (6, 8), (7, 8), (7, 7), (7, 8), (6, 8), (7, 8), (6, 8), (6, 7), (6, 8), (7, 8), (7, 9), (7, 8), (7, 7), (7, 6), (7, 7), (7, 8), (8, 8), (8, 7), (7, 7), (6, 7), (5, 7), (5, 6), (4, 6), (5, 6), (6, 6), (7, 6), (8, 6), (8, 7), (8, 8), (8, 9), (9, 9), (9, 8), (10, 8), (9, 8), (10, 8), (10, 9), (11, 9), (11, 8), (12, 8), (12, 9), (12, 10), (11, 10), (11, 11), (12, 11), (12, 12), (11, 12), (11, 11), (11, 10), (11, 11), (11, 12), (11, 11), (12, 11), (12, 12), (11, 12), (10, 12), (9, 12), (8, 12), (9, 12), (8, 12), (9, 12), (9, 11), (8, 11), (8, 12), (7, 12), (6, 12), (6, 11), (7, 11), (7, 12), (8, 12), (8, 11), (8, 10), (9, 10), (10, 10), (11, 10), (10, 10), (11, 10), (11, 11), (10, 11), (10, 10), (9, 10), (8, 10), (9, 10), (9, 9), (9, 8), (10, 8), (11, 8), (10, 8), (10, 7), (11, 7), (11, 6), (11, 5), (11, 4), (12, 4), (12, 3), (12, 2), (12, 1), (11, 1), (10, 1), (9, 1), (8, 1), (8, 2), (9, 2), (8, 2), (8, 3), (7, 3), (7, 2), (8, 2), (8, 1), (8, 2), (9, 2), (9, 3), (8, 3), (8, 2), (9, 2), (8, 2), (8, 1), (8, 2), (8, 3), (7, 3), (7, 4), (8, 4), (9, 4), (9, 3), (9, 4), (8, 4), (9, 4), (10, 4), (10, 3), (10, 2), (10, 3), (10, 4), (11, 4), (12, 4), (12, 5), (12, 6), (12, 5), (11, 5), (11, 6), (11, 5), (11, 4), (11, 3), (11, 4), (11, 5), (10, 5), (11, 5), (12, 5), (11, 5), (11, 4), (11, 5), (10, 5), (10, 6), (11, 6), (11, 5), (11, 6), (11, 5), (12, 5), (12, 4), (12, 3), (12, 4), (12, 3), (12, 2), (12, 1), (12, 2), (12, 1), (11, 1), (12, 1), (11, 1), (11, 2), (11, 1), (10, 1), (11, 1), (10, 1), (10, 2), (10, 3), (9, 3), (8, 3), (9, 3), (8, 3), (8, 2), (8, 1), (7, 1), [8, 1]]

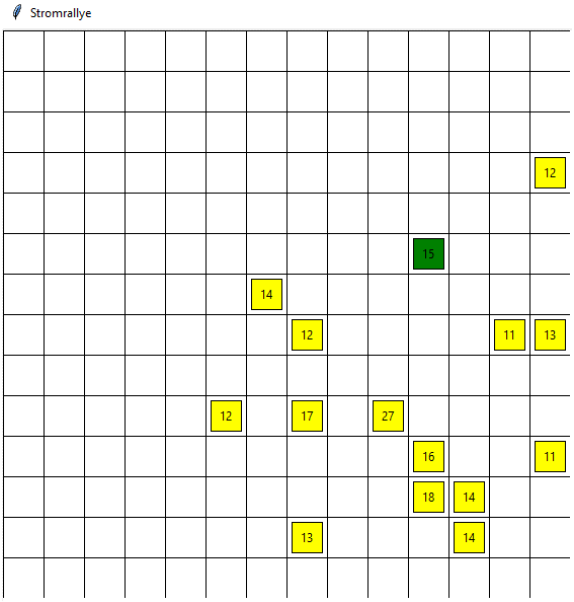
(Die Abfolge der Schritte in deutscher Sprache wurde hier ebenfalls weggelassen.)

Schwierigkeitsgrad schwer mit 14x14 Spielfeld:

>> Laufzeit des Programms: 0.0019999378204345703 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

>> Spielgröße: 14
>> Startposition des Roboters (11, 6, 15)
>> Anzahl der Batterien: 14
>> Koordinaten der Batterien: [(8, 13, 13), (11, 12, 18), (10, 10, 27), (11, 11, 16), (8, 8, 12), (12, 12, 14), (14, 11, 11), (12, 13, 14), (8, 10, 17), (7, 7, 14), (6, 10, 12), (13, 8, 11), (14, 8, 13), (14, 4, 12)]

>> Ausgabe im BwInf-Format:
14
11,6,15
14
8,13,13
11,12,18
10,10,27
11,11,16
8,8,12
12,12,14
14,11,11
12,13,14
8,10,17
7,7,14
6,10,12
13,8,11
14,8,13
14,4,12



Screenshot der erzeugten Spielsituation

> Lösung:
>> Alle besuchten Felder in richtiger Reihenfolge:
> [(11, 6), (12, 6), (12, 7), (13, 7), (12, 7), (12, 6), (13, 6), (13, 5), (13, 4), (12, 4), (12, 5), (12, 6), (12, 5), (13, 5), (14, 5), (14, 4), (13, 4), (13, 5), (13, 6), (13, 7), (14, 7), (13, 7), (13, 6), (14, 6), (13, 6), (13, 7), (14, 7), (14, 8), (14, 7), (14, 6), (14, 5), (13, 5), (14, 5), (13, 5), (13, 6), (12, 6), (13, 6), (12, 6), (13, 6), (13, 7), (13, 8), (12, 8), (11, 8), (10, 8), (9, 8), (9, 9), (9, 10), (9, 9), (8, 9), (7, 9), (6, 9), (6, 10), (7, 10), (7, 9), (8, 9), (7, 9), (6, 9), (7, 9), (7, 8), (6, 8), (7, 8), (7, 9), (7, 8), (7, 7), (7, 8), (7, 9), (7, 8), (7, 9), (7, 10), (7, 11), (8, 11), (9, 11), (8, 11), (9, 11), (8, 11), (8, 12), (8, 11), (8, 10), (8, 11), (9, 11), (9, 12), (9, 11), (9, 12), (10, 12), (10, 11), (10, 12), (10, 11), (10, 12), (10, 11), (10, 12), (10, 13), (10, 14), (11, 14), (11, 13), (12, 13), (12, 14), (13, 14), (14, 14), (14, 13), (14, 14), (13, 14), (14, 14), (14, 13), (13, 13), (13, 14), (14, 14), (14, 13), (14, 12), (14, 11), (14, 10), (13, 10), (12, 10), (11, 10), (12, 10), (12, 11), (12, 10), (12, 11), (12, 10), (12, 11), (12, 12), (12, 11), (13, 11), (13, 10), (13, 9), (12, 9), (12, 8), (11, 8), (11, 7), (11, 8), (10, 8), (9, 8), (9, 9), (8, 9), (8, 8), (9, 8), (10, 8), (10, 9), (9, 9), (9, 8), (10, 8), (10, 9), (11, 9), (10, 9), (11, 9), (11, 10), (11, 11), (11, 10), (12, 10), (12, 9), (13, 9), (14, 9), (14, 10), (13, 10), (12, 10), (12, 9), (12, 8), (12, 9), (11, 9), (12, 9), (11, 9), (11, 10), (10, 10), (10, 11), (10, 12), (10, 11), (9, 11), (10, 11), (10, 12), (10, 13), (9, 13), (9, 14), (8, 14), (7, 14), (7, 13), (7, 12), (7, 13), (7, 14), (8, 14), (7, 14), (8, 14), (9, 14), (10, 14), (10, 13), (10, 12), (9, 12), (10, 12), (10, 11), (10, 12), (11, 12), (10, 12), (9, 12), (9, 13), (10, 13), (10, 12), (9, 12), (9, 11), (9, 12), (10, 12), (10, 11), (9, 11), (9, 12), (9, 13), (9, 12), (9, 13), (10, 13), (9, 13), (8, 13), (7, 13), (7, 14), (7, 13), (6, 13), (6, 12), (6, 11), (7, 11), (8, 11), (9, 11), (8, 11), (7, 11), (7, 12), [6, 12]]

(Hier wurde auch aus platztechnischen Gründen die Ausgabe der Schrittanweisungen in deutscher Sprache weggelassen.)

Quellcode

```
class SpielErzeugen:
    def __init__(self, schwierigkeit: int, größe_spielfeld: int):
        """  Erstellen einer Spielsituation

        Schwierigkeit ist im Bereich von 1 bis 3:
            - 1 --> leicht
            - 2 --> mittel
            - 3 --> schwer

        Größen des Spielfelds
            - 10 --> 10 x 10
            - 12 --> 12 x 12
            - 14 --> 14 x 14

        """
        # Start der Zeitmessung
        self.start_zeit = time.time()

        # Klassenvariablen werden zugewiesen
        self.schwierigkeit = schwierigkeit
        self.size = größe_spielfeld


        # Zufallszahlen für die x- und y-Koordinate des Startpunktes
        # diese liegen im Bereich 1 bis Größe des Spielfelds
        x_start = randint(1, self.size)
        y_start = randint(1, self.size)

        startpunkt = [x_start, y_start]

        # Bereich der Schritte festlegen
        if self.schwierigkeit == 1:
            self.bereich_schritte = [1, 5]
        elif self.schwierigkeit == 2:
            self.bereich_schritte = [5, 10]
        elif self.schwierigkeit == 3:
            self.bereich_schritte = [10, 15]
        else:
            raise ValueError

        # Anzahl Batterien festlegen (Anzahl Züge)
        if self.schwierigkeit == 1:
            self.anzahl_batterien = 5
        elif self.schwierigkeit == 2:
            self.anzahl_batterien = 10
        elif self.schwierigkeit == 3:
            self.anzahl_batterien = 15
        else:
            raise ValueError

        # Speicherung der Batterien
        self.batterien = []
```

```
        # aktueller Punkt wird auf den zufällig ausgewählten Startpunkt gesetzt
        aktueller_punkt = startpunkt.copy()

        # in gesamte_positionen werden alle nacheinander besuchten Felder gespeich
ert
        gesamte_positionen = [aktueller_punkt]

        # für jede Batterie:
        for a in range(self.anzahl_batterien):

            # x- und y-
Koordinaten der bisher gespeicherten Batterien wird ermittelt
            batterien_x_y = list(map(
                lambda batterie: batterie[:2], self.batterien
            ))

            # Anzahl der Schritte wird zufällig bestimmt
            anzahl_schritte = randint(*self.bereich_schritte)

            # besuchte Felder bevor eine neue Batterie 'gelegt' wird
            positionen = []

            # mithilfe von s wird die while-Schleife gesteuert
            s = anzahl_schritte
            while s >= 0:

                # falls alle Schritte gemacht wurden
                # und die Batterie im nächsten Schritt gelegt wird,
                # wird überprüft, ob dieses Feld nicht schon mal durch Schritte be
reits besucht wurde
                # Ist dies der Fall so wird ein noch ein weiterer Schritt gemacht
                if s == 0:
                    if aktueller_punkt in gesamte_positionen or aktueller_punkt in
 positionen[:-1]:
                        anzahl_schritte += 1
                        s += 1
                    else:
                        # wenn nicht, dann wird s um 1 verringert und die Batterie
 anschließend gelegt
                        s -= 1
                    continue

                # eine neue Position (neues Feld) wird bestimmt
                neue_position = self.randomStep(*aktueller_punkt, batterien_x_y)

                # falls keine neue Position gefunden wurde,
                # wird noch mal 'zurückgegangen' und ein anderes Feld ausgewählt
                if neue_position == None:
                    positionen.pop()
                    s += 1
                else:
                    positionen.append(neue_position)
                    s -= 1


                # aktueller Punkt ist das letzte hinzugefügte Feld
                if positionen:
                    aktueller_punkt = positionen[-1]
                else:
```

```
                    # falls in positionen keine Felder besucht wurden,
                    # wird das letzte Element von gesamte_positionen ausgewählt
                    aktueller_punkt = gesamte_positionen[-1]

            # Auf dem aktuellen Feld (aktueller Punkt) wird eine Batterie gelegt.
            # Die Ladung der Batterie ist die Anzahl der zurückgelegten Schritte
            self.batterien.append(
                (*aktueller_punkt, anzahl_schritte)
            )

            # die gesamte Liste der besuchten Felder wird erweitert
            gesamte_positionen.extend(positionen)

        # die besuchten Felder werden umgedreht,
        # da der Roboter 'von hinten' starten soll
        gesamte_positionen.reverse()
        self.gesamte_positionen = gesamte_positionen

        # Spielfeld wird graphisch erzeugt
        self.erzeugeUmgebung()

    def randomStep(self, x_akt: int, y_akt: int, vorhandene_batterien: list):
        """ Methode zum Auswählen eines zufälligen Nachbarfeldes, das besucht werd
en kann.

        Das zu besuchende Nachbarfeld sollte nicht durch eine Ersatzbatterie berei
ts belegt sein.

        Args:
            x_akt (int): x-Koordinate des Ausgangsfelds
            y_akt (int): y-Koordinate des Ausgangsfelds
            vorhandene_batterien (list): Liste mit den x- und y-
Koordinaten der bisher erstellten Batterien

        Returns:
            tuple. Nachbarfeld, das frei ist und somit besucht werden kann
        """

        # in schritte werden die möglichen Nachbarfelder hinzugefügt
        schritte = []
        aktuelle_position = [x_akt, y_akt]

        # Die Positionen der Felder unterhalb, oberhalb, rechts und links des Ausg
angsfeld werden bestimmt
        # Diese werden als tuple gespeichert.

        unten = aktuelle_position.copy()
        unten[1] += 1
        unten = tuple(unten)
        if unten in vorhandene_batterien:
            unten = None

        oben = aktuelle_position.copy()
        oben[1] -= 1
        oben = tuple(oben)
        if oben in vorhandene_batterien:
            oben = None

        rechts = aktuelle_position.copy()
```

```
        rechts[0] += 1
        rechts = tuple(rechts)
        if rechts in vorhandene_batterien:
            rechts = None

        links = aktuelle_position.copy()
        links[0] -= 1
        links = tuple(links)
        if links in vorhandene_batterien:
            links = None

        # Je nach Lage der aktuellen Position werden die Nachbarfelder zu den mögl
ichen Schritten hinzugefügt
        if y_akt > 1:
            schritte.append(oben)

        if y_akt < self.size:
            schritte.append(unten)

        if x_akt > 1:
            schritte.append(links)

        if x_akt < self.size:
            schritte.append(rechts)

        # Alle 'None' werden entfernt
        [schritte.remove(None) for i in range(schritte.count(None))]


        # falls mögliche Nachbarfeld gefunden wurden
        # und somit ein Schritt möglich ist
        if schritte:
            # auszuwählender Bereich wird bestimmt
            bereich = len(schritte)-1

            # ein zufälliger Index im Bereich wird ausgewählt
            zufalls_index = randint(0, bereich)

            # über den Index wird das zufällige Nachbarfeld ausgewählt
            # und somit der Zufallsschritt bestimmt
            zufallsschritt = schritte[zufalls_index]

            return zufallsschritt


        # falls nicht, wird None zurückgegeben
        else:
            return None

    def erzeugeUmgebung(self):
        """ Methode zur Erstellung der graphischen Benutzeroberfläche mithilfe der
 Klasse Environment """

        # der Startpunkt des Roboters ist die letzte Batterie
        roboter = self.batterien.pop(-1)
        anzahl_batterien = len(self.batterien)

        self.ende_zeit = time.time()
        benötigte_zeit = self.ende_zeit - self.start_zeit
```

```
        print(f">> Laufzeit des Programms: {benötigte_zeit} Sekunden \n (Start der
 Zeitmessung bei Aufruf der main-Methode)\n\n")


        # Konsolenausgabe mit den Daten
        print(f">> Spielgröße: {self.size}")
        print(f">> Startposition des Roboters {roboter}")
        print(f">> Anzahl der Batterien: {anzahl_batterien}")
        print(f">> Koordinaten der Batterien: {self.batterien}")

        # Ausgabe im BwInf-Format
        print(f"\n >> Ausgabe im BwInf-Format: \n{self.size}")
        print(f"{roboter[0]},{roboter[1]},{roboter[2]}")
        print(f"{anzahl_batterien}")
        for batterie in self.batterien:
            print(f"{batterie[0]},{batterie[1]},{batterie[2]}")

        # Visualisierung wird erstellt
        self.umgebung = Environment(self.size, roboter, len(self.batterien), self.
batterien)

    def wegInAnweisungen(self, punkte: list):
        """ Methode zur Konvertierung von einzelnen Punkten in Schritte

        Args:
            punkte (list): Liste mit Punkten, aus denen eine Abfolge von Schritten
 ermittelt werden soll.

        Returns:
            list. Abfolge von Schrittanweisungen für den Roboter, z.B. 0 --
> Schritt nach oben
        """

        # in abfolge_schritte werden alle Schrittanweisungen gespeichert
        abfolge_schritte = []

        for index in range(len(punkte)):
            if index > 0:
                # Teilweg aus den Punkten P1 und P2
                p1 = punkte[index-1]
                p2 = punkte[index]

                # die Differenzen der x- und y-Koordinaten wird berechnet
                delta_x = p2[0] - p1[0]
                delta_y = p2[1] - p1[1]

                # je nach Differenz muss eine bestimmte Bewegung ausgeführt werden
,
                # um von P1 zu P2 zu gelangen

                # nach rechts
                if delta_x > 0:
                    bewegung = 3

                # nach links
                elif delta_x < 0:
                    bewegung = 2

                # nach unten
```

```
                elif delta_y > 0:
                    bewegung = 1

                # nach oben
                elif delta_y < 0:
                    bewegung = 0

                abfolge_schritte.append(bewegung)

        return abfolge_schritte

    def zeigeLösung(self):
        """ Methode zum Ausgeben der Lösung in der Konsole und Visualisierung der
Schritte

        Diese Methode kann bei Bedarf vom Eingabefenster aufgerufen werden, wenn d
er Benutzer eine Lösung sehen möchte.
        """

        # Konvertierung der einzelnen besuchten Felder in Schrittanweisungen für d
en Roboter
        schrittanweisungen_roboter_lösung = self.wegInAnweisungen(self.gesamte_pos
itionen)

        print(f"\n\n> Lösung: \n>> Alle besuchten Felder in richtiger Reihenfolge:
 \n > {self.gesamte_positionen}")

        # Abfolge der Schritte wird von Zahlen zu deutschen Wörtern 'konvertiert'
        abfolge_schritte_deutsch = []
        for schritt in schrittanweisungen_roboter_lösung:
            if schritt == 0:
                abfolge_schritte_deutsch.append('oben')
            elif schritt == 1:
                abfolge_schritte_deutsch.append('unten')
            elif schritt == 2:
                abfolge_schritte_deutsch.append('links')
            elif schritt == 3:
                abfolge_schritte_deutsch.append('rechts')

        print(f"\n>> Abfolge von {len(abfolge_schritte_deutsch)} Schritten für den
 Roboter in deutscher Sprache: \n > {abfolge_schritte_deutsch}")

        # Roboter wird mithilfe der Schrittanweisungen gesteuert
        for schritt in schrittanweisungen_roboter_lösung:
            self.umgebung.step(schritt)
            self.umgebung.update()

        tk.mainloop()
```