Aufgabe 1: Stromrallye

Teilnahme-ID: 52570

Bearbeiter dieser Aufgabe:

Florian Rädiker (16 Jahre)

18. April 2020

Inhalt

1 Lösungsidee

Um einen Weg für den Roboter finden zu können, ist es sinnvoll, die Spielsituation in einen Graphen umzuwandeln und in diesem dann nach einem Weg zu suchen. Bei der Umwandlung wird auf den wichtigen Umstand zurückgegriffen, dass immer nur auf den Feldern eine Batterie liegen kann, auf denen sich zu Beginn eine Ersatzbatterie befand. Das liegt daran, dass der Roboter eine Batterie nur ablegen kann, wenn er ein Feld mit einer Batterie betritt – also ein Feld, auf dem zu Beginn eine Ersatzbatterie lag. Diese Felder heißen *Batteriefelder*.

1.1 Grundüberlegungen zur Bildung eines Graphen

Damit eine Batterie, die der Roboter auf einem Batteriefeld soeben aufgenommen hat, leer wird und der Roboter sich trotzdem weiterbewegen kann, sollte er sich so zu einem Batteriefeld bewegen, auf dem eine noch geladene Batterie liegt, dass seine Bordbatterie beim Betreten des Feldes die Ladung 0 erreicht. Um dies zu erreichen, muss es einen Weg zwischen dem aktuellen Feld und dem anderen Batteriefeld geben, dessen Länge genauso groß ist wie die Ladung der neu aufgenommenen Batterie. Der Weg muss nicht der kürzeste zwischen den beiden Feldern sein, sondern es ist auch möglich, Batterieladung „verfallen“ zu lassen. Sobald der kürzeste Weg zwischen zwei Batteriefeldern länger als zwei Felder ist, hat der Roboter immer die Möglichkeit, Ladung verfallen zu lassen, weil sich dann zwischen den beiden Feldern mindestens zwei freie Felder befinden, auf denen der Roboter hin- und herlaufen kann. In Abb. 1 gibt es einen Weg zwischen den Feldern A und B der Länge 2. Nimmt der Roboter diesen Weg, kann er keine Ladung verfallen lassen, weil es keine Möglichkeit für ihn gibt, hin- und herzulaufen. Es ist auch nicht möglich, von Feld A aus einen Schritt nach rechts zu gehen und dann wieder zurück auf Feld A – dann würde der Roboter die dort liegende Batterie mit Ladung 0 aufnehmen müssen. Es existiert jedoch auch ein Weg der Länge 6 zwischen den Feldern A und B. Diesen Weg könnte der Roboter aber nur nehmen, wenn auf Feld A nicht eine Batterie mit Ladung 4, sondern mindestens mit Ladung 6 läge. Für die in Abb. 1 gezeigte Spielsituation existiert keine Lösung. Etwas anders sähe die Situation aus, wenn es sich bei Feld A um das Startfeld des Roboters handelt. Das Startfeld des Roboters wird auch als Batteriefeld angesehen, allerdings „blockiert“ es den Weg für den Roboter nicht wie andere Batteriefelder, weil auf dem Startfeld nie eine Batterie liegt. Daher würde in diesem Fall auch ein Weg der Länge 4 von A nach B existieren, weil der Roboter auf Feld A zurückkehren kann, ohne dort eine Batterie aufnehmen zu müssen.

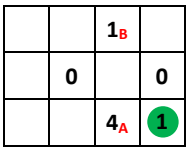


Abb. 1. Nach dem ersten Schritt befindet sich der Roboter auf Feld A, auf das er eine Batterie mit Ladung 0 ablegt und von dem er eine Batterie mit Ladung 4 aufnimmt.

Eine weitere Voraussetzung dafür, dass der Roboter mit Ladung 0 auf einem anderen Feld ankommen kann, ist, dass die Distanz dieselbe Parität haben muss wie die Ladung der aufgenommenen Batterie, weil sich der Roboter für jedes Feld, um das er sich gegebenenfalls vom Zielfeld entfernt, wieder ein Feld in Richtung des Zielfelds bewegen muss. Der Roboter kann also nur eine gerade Ladung verfallen lassen.

Die Bordbatterie muss nicht immer die Ladung 0 haben, wenn der Roboter auf einem Batteriefeld ankommt und Batterien tauscht. Der Roboter kann eine Batterie auch auf einem Feld erstmal ablegen und später, wenn er erneut auf das Feld gelangt, sie wieder aufnehmen. Insbesondere ist es auch möglich, dass der Roboter eine Batterie von Feld X aufnimmt und mit dieser Batterie zu Feld X zurückkehrt.

Sobald alle Batterien außer der Bordbatterie entladen sind, muss der Roboter auf kein Batteriefeld gelangen, sondern er darf auf einem beliebigen Feld „liegenbleiben“. Nicht jedes Batteriefeld ist dafür geeignet, das letzte Feld zu sein. Nimmt der Roboter als letzte eine Batterie mit Ladung 1 auf, gibt es keine besonderen Anforderungen an das Batteriefeld:[1] Der Roboter bewegt sich einfach auf ein

[1] Außer wenn das Spielbrett nur aus einem Feld besteht, auf dem der Roboter startet. In diesem Fall gibt es keine Lösung.

beliebiges benachbartes Feld (Abb. 2). Dabei ist es irrelevant, ob auf dem Feld eine Batterie liegt oder nicht, denn in jedem Fall sind nun alle Batterien leer. Anders sieht es aus, wenn der Roboter zuletzt auf ein Feld gelangt, auf dem sich eine Batterie mit der Ladung 2 befindet. Dann muss sich neben diesem Batteriefeld (nicht diagonal) ein freies Feld befinden, auf das sich der Roboter bewegen kann. Befinden sich neben dem Batteriefeld zwei zusammenhängende freie Felder, dann ist jede Batterieladung möglich: Der Roboter kann sich auf den beiden Feldern solange bewegen, bis die Batterie leer ist.

	0	
0	**Z**	**0**
	1	**1**

Abb. 2. Für diese Spielsituation existiert nur eine Lösung, wenn auf Feld Z eine Batterie mit Ladung 0 oder 1 liegt. Mit größerer Ladung ist es für den Roboter nicht möglich, die restliche Ladung zu „verfahren".

Zusammenfassend ist für den Graphen und dessen Knoten, die Batteriefelder, nur relevant,

- wie lang der kürzeste Weg zwischen zwei Batteriefeldern ist,
- falls der kürzeste Weg zwischen zwei Batteriefeldern 1 oder 2 Felder lang ist: Ob es überhaupt Wege mit mindestens Länge 3 gibt und wenn ja, wie lang der kürzeste dieser Wege ist (in Abb. 1 wäre dies Länge 6 für die Batteriefelder A und B)[2],
- welche Ladung die Batterie besitzt, die sich auf dem Batteriefeld zu Beginn befindet, und
- ob sich keines, eines oder mindestens zwei zusammenhängende freie Felder neben einem Batteriefeld befinden.

1.2 Umwandlung der Spielsituation in einen Graphen

1.2.1 Gewichteter Digraph: Entfernungen zwischen den Batterien

Um eine Spielsituation in einen Graphen umzuwandeln, werden zunächst die kleinstmöglichen Distanzen zwischen allen Batteriefeldern bestimmt. Es reicht dabei nicht aus, für je zwei Batteriefelder die Manhattan-Distanz zu bestimmen. Die Distanz zwischen zwei Batteriefeldern kann nie kleiner als die Manhattan-Distanz sein, es ist jedoch möglich, dass die Distanz größer ist oder ein Batteriefeld von einem anderen aus gar nicht erreicht werden kann. Die Distanz wird größer oder ein Batteriefeld kann nicht erreicht werden, wenn andere Batteriefelder den Weg „versperren". Da der Roboter seine Batterie wechseln muss, sobald er ein Batteriefeld betritt, darf er auf dem Weg von einem Batteriefeld zu einem anderen nur freie Felder einschlich des Roboterstartfelds betreten.

Die Distanzen werden folgendermaßen bestimmt: b sei ein Array der Länge l mit allen Batteriefeldern inklusive Roboterstartfeld. Wenn die Distanz von A nach B gefunden wurde, muss keine Distanz von B nach A gesucht werden – die Distanz ist dieselbe. Von jedem Batteriefeld aus b werden die Distanzen zu allen Batteriefeldern betrachtet, die in b weiter rechts stehen, also einen größeren Index haben. Die Distanzen von einem Batteriefeld aus werden mit dem Lee-Algorithmus [1] bestimmt. Beim Lee-Algorithmus handelt es sich um ein auf Breitensuche basierendes Maze-Routing-Verfahren. Das Startfeld, hier das Batteriefeld, von dem aus die Distanzen berechnet werden sollen, wird mit Distanz 0 markiert. Alle freien (d.h. sie sind keine Batteriefelder oder das Roboterstartfeld) Nachbarfelder des Startfelds erhalten die Distanz 1, und alle freien, noch nicht markierten Nachbarfelder der Nachbarfelder die Distanz 2 und so weiter. Auf diese Weise werden die Felder in einer Art „Wellen" markiert. Das Maze-Routing wird beendet, sobald die Distanz zu allen anderen Batteriefeldern (denen, die sich in b weiter rechts befinden) gesetzt wurde. Um Wege zwischen den Batteriefeldern speichern zu können, kann ausgehend von einem Batteriefeld der Weg zum Startfeld zurückverfolgt werden.

[2] Es reicht nicht aus, einen beliebigen Weg länger als 2 zu speichern, sondern es muss der kürzeste sein, weil ansonsten für einige Batterieladungen kein Weg gefunden würde. Würde in Abb. 1 zwischen A und B für den Weg länger als 2 Felder ein Weg mit Länge 8 gespeichert, dann wäre nicht bekannt, dass der Roboter auch mit einer Batterie mit Ladung 6 von A nach B gelangen kann.

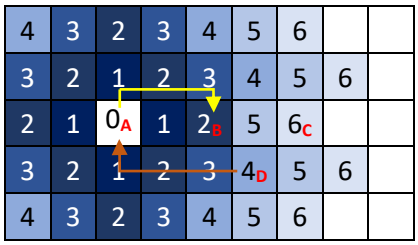


Abb. 3. Anwendung des Lee-Algorithmus. Ausgehend von Feld A mit Distanz 0 werden Wege zu den Feldern B, C und D gesucht. Ab der Distanz 6 sind Entfernungen zu allen Feldern bekannt, also müssen keine weiteren Felder markiert werden.

In Abb. 3 ist A das Startfeld. Ein Weg von A zu D kann ermittelt werden, indem, von D ausgehend, jeweils das Nachbarfeld mit Distanz = *Distanz des aktuellen Feldes* − 1 besucht wird, solange, bis die Distanz 0 ist, also das Startfeld erreicht wurde. Dieser Weg rückwärts ist ein möglicher kürzester Weg von A nach D. Von D ist das nächste Nachbarfeld mit einer um 1 kleineren Distanz das Feld zur linken Seite (Distanz 3). Vom Feld mit Distanz 3 gibt es zwei mögliche Wege, nach links oder oben. Einer der möglichen Wege ist in orange eingezeichnet.

Mit dem Maze-Routing-Verfahren ist es auch möglich, den kürzesten Weg der Wege länger als 2 Felder zu bestimmen. In Abb. 3 ist der Weg zwischen A und B zwei Felder lang, weshalb ein längerer Weg gefunden werden muss. Sobald der kürzeste Weg zu Feld B gefunden wurde, wird festgestellt, dass ein längerer Weg benötigt wird und Feld B wird als ein Feld markiert, für das ein längerer Weg gefunden werden muss. In Abb. 3 wird beim Markieren des Felds über oder unter Feld B mit Distanz 3 festgestellt, dass dieses Feld ein Nachbarfeld hat, für das ein längerer Weg gesucht wird. Für Feld B wurde damit ein längerer Weg gefunden, dieser hat die Distanz 3+1=4. Ein möglicher dieser Wege ist in gelb eingezeichnet.

Alle Nachbarfelder von regulär besuchten Feldern auf die beschriebene Weise zu überprüfen, funktioniert nicht, wenn der Roboter für einen längeren Weg ein Feld doppelt betreten muss. Dieser Fall tritt in dem Spielbrett auf, das Abb. 4 zeigt.

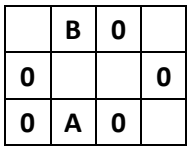


Abb. 4. Ein Spielbrett, in dem ein Weg länger als zwei Felder zwischen den Batteriefeldern A und B gefunden werden muss. Auf den mit „0“ beschrifteten Feldern befinden sich andere Batterien mit irrelevanter Ladung.

Um dies zu berücksichtigen, wird das Maze-Routing-Verfahren so abgewandelt, dass Felder doppelt markiert werden können. Die Distanz, mit der ein Feld zuerst markiert wird, ist immer die kleinste und damit der kürzeste Weg zu diesem Feld. Die zweite Distanz stellt einen nicht-kürzesten Weg zu diesem Feld dar. Ein Feld muss maximal zweimal mit einer Distanz markiert werden, weil beim zweiten Markieren eines Feldes der Weg zu diesem Feld auf jeden Fall länger als zwei Felder ist und der Roboter somit die Möglichkeit hat, „hin- und herzulaufen“.

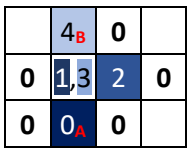


Abb. 5. Anwendung des erweiterten Maze-Routing-Verfahrens auf das Spielbrett aus Abb. 4. Nach dem Besuchen des Felds mit Distanz 2 wird das vorher mit 1 markierte Feld nochmals besucht und nun zusätzlich mit 3 markiert. Vom mit „1,3“ markierten Feld aus werden zwei Wege zu Feld B gefunden: Einmal, als das Feld mit Distanz 1 markiert wurde, und einmal, als das Feld mit Distanz 3 markiert wurde.

Ab einer Distanz von 4 ist es nicht mehr nötig, dass Felder doppelt markiert werden oder bereits gesetzte zweite Distanzen für weitere Wege berücksichtigt werden. Würde in einem Weg mit Länge 5 ein Feld doppelt besucht, so gäbe es einen kürzeren Weg der Länge 3. Dieser kürzere Weg lässt sich auch finden, ohne dass Felder doppelt besucht werden:

	5_B	
0	4	**0**
0	1,3	2
	0_A	

	3_B	
0	2	**0**
0	1	
	0_A	

Abb. 6. Beispiel dafür, dass ab einer Länge von 5 keine Felder mehrfach besucht werden müssen. Links der längere Weg mit Länge 5, bei dem ein Feld doppelt besucht wird. Ohne ein Feld doppelt zu besuchen, kann Feld B aber noch kürzer erreicht werden, und der Weg ist trotzdem länger als 2 Felder. (Im Beispiel müsste eigentlich kein Weg länger als 2 gesucht werden, weil der kürzeste Weg schon 3 Felder lang ist. Es soll lediglich illustriert werden, dass ab einer Länge von 5 keine Felder doppelt besucht werden müssen.)

Die Anwendung des Maze-Routing-Verfahrens ist recht aufwändig. Zum einen muss eine steigende Anzahl von Feldern, die als nächstes besucht werden sollen, in der Warteschlange für die Breitensuche gespeichert werden, zum anderen werden viele Felder besucht, die für die Wege nicht relevant sind. Wenn Distanzen von einem Batteriefeld zu anderen Batteriefeldern (den Batteriefeldern, die in b einen größeren Index haben) berechnet werden, wird aus diesen Gründen versucht, die Zahl der Batteriefelder zu reduzieren, zu denen die Distanz per Maze-Routing bestimmt werden muss. Ist die Distanz zu einem anderen Batteriefeld kleiner als 3, muss für die Bestimmung der längeren Distanz immer Maze-Routing eingesetzt werden. Für jedes Batteriefeld wird überprüft, ob ein einfacher Weg zu diesem Batteriefeld mit der Länge der Manhattan-Distanz frei ist, sich also auf dem Weg keine Ersatzbatterien befinden. Dazu werden die Batteriefelder in b nach Position sortiert, die y-Position zählt dabei mehr als die x-Position (das Sortieren ist kein notwendiger Schritt, es sorgt allerdings für mehr Übersichtlichkeit). In Abb. 7 besteht b aus den Batteriefeldern A, B, C, D, E, F und G, in dieser Reihenfolge. Sollen Distanzen von Feld C aus bestimmt werden, wurden bereits die Distanzen AB und AC bestimmt. Von C aus werden daher nur die Distanzen ab D betrachtet. Die x-Position des aktuellen Feldes, hier C, sei x_1; die y-Position y_1. Die Position des Feldes, zu dem eine Distanz bestimmt werden soll (hier D, E, F oder G), sei (x_2, y_2). Es gilt $y_1 \leq y_2$. Der einfache Weg, der nun betrachtet wird, verläuft zunächst horizontal von x_1 zu x_2 auf der Höhe y_1, danach vertikal bis y_2.[3] Auf allen Feldern (x, y) mit $y = y_1$ und $\min(x_1, x_2) < x < \max(x_1, x_2)$ sollten sich also keine Ersatzbatterien befinden, damit der Weg frei ist. Dasselbe gilt für Felder (x, y) mit $x = x_2$ und $y_1 < y < y_2$. Außerdem gibt es ein „Eckfeld“ bei (x_2, y_1), wenn $x_1 \neq x_2 \wedge y_1 \neq y_2$, auf dem ebenfalls keine Ersatzbatterie liegen darf. Sind alle diese Felder frei, ist die Distanz zwischen den beiden Feldern $|x_1 - x_2| + (y_2 - y_1)$ und es muss kein Maze-Routing durchgeführt werden, um die Distanz zum Feld (x_2, y_2) zu bestimmen, außer die Distanz ist 1 oder 2. In Abb. 7 muss auf diese Weise von Feld C aus nur Maze-Routing durchgeführt werden, um die Distanz zwischen C und G zu ermitteln und um für CD und CF Wege zu finden, die länger als 2 Felder sind.

	A			
B		**C**		
			D	
	E	**F**	**G**	

Abb. 7.

Das Roboterstartfeld bedarf einer besonderen Betrachtung bei der Ermittlung der Distanzen. Das Roboterstartfeld wird immer als freies Feld angesehen, weil sich auf diesem Feld nie eine Batterie befindet. Dennoch werden wie für andere Felder auch Distanzen zwischen dem Roboterstartfeld und anderen Feldern bestimmt. Diese Distanzen gelten jedoch immer nur vom Roboterstartfeld weg, da es nicht

[3] Es wird nur dieser eine Weg betrachtet. Der zweite Weg dieser Art, der zuerst vertikal und dann horizontal verläuft, könnte auch betrachtet werden, um die Anzahl der Batterien, für die Maze-Routing durchgeführt werden muss, noch weiter zu reduzieren.

möglich ist, zum Roboterstartfeld zurückzukehren und dort eine Batterie aufzunehmen. Aus diesem Grund ist der entstehende Graph gerichtet. Kanten, die das Roboterstartfeld mit einem anderen Feld verbinden, führen immer vom Roboterstartfeld weg. Für jede andere Kante existiert eine gegenläufige Kante.

Die zusammenhängenden freien Felder neben einem Batteriefeld können mit dem Maze-Routing-Verfahren bestimmt werden. Gibt es ein mit 2 (als erste Distanz) markiertes Feld, gibt es zwei zusammenhängende freie Felder, wurden nur Felder mit 1 markiert, gibt es nur eines; ansonsten gibt es keine.

Wie eingangs beschrieben, ist es auch möglich, dass der Roboter eine Batterie von Feld X aufnimmt, die Ladung um eine gerade Zahl reduziert, und wieder zu Feld X zurückkehrt. Die Ladung kann um zwei reduziert werden, wenn Feld X ein freies Nachbarfeld hat. Hat Feld X zwei zusammenhängende freie Felder, kann die Batterieladung um eine beliebige gerade Zahl reduziert werden. Im Graphen wird dies berücksichtigt, indem entsprechende Schleifen existieren. Die Schleifen stellen immer eine Distanz von 2 dar und haben, falls das Batteriefeld zwei zusammenhängende freie Felder besitzt, einen Weg länger als 2 mit der Länge 4. Schleifen werden nicht am Roboterstartfeld gebildet.

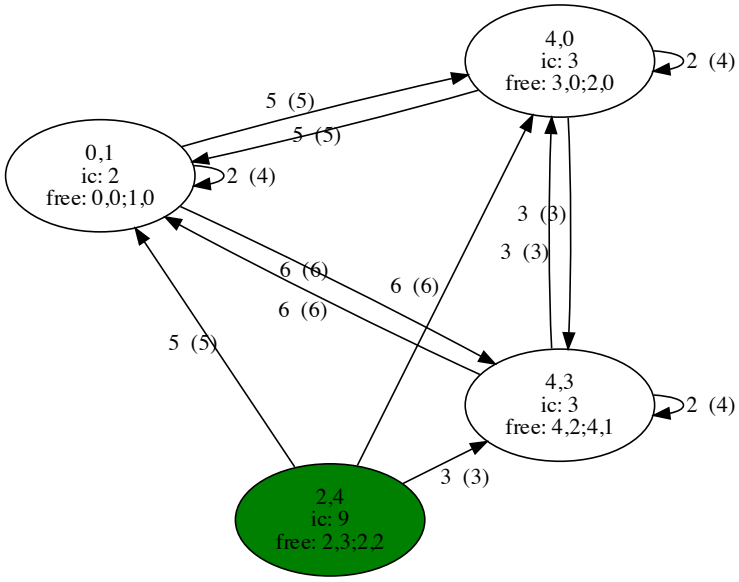


Abb. 8. Der für stromrallye0.txt entstandene Graph. „ic“ steht für „initial charge“, also die Ladung der Batterie, die sich zu Beginn auf diesem Feld befindet. Unter „free“ sind die Positionen der zusammenhängenden freien Felder aufgeführt. Für jedes Feld gibt es zwei solcher zusammenhängender freier Felder, weshalb sich jedes Batteriefeld mit beliebiger Ladung dazu eignet, als letztes besucht zu werden. In Klammern steht an jeder Kante, welche Länge ein Weg zwischen den Batteriefeldern hat, der länger als 2 Felder ist. Die Positionen haben den Ursprung (0, 0), nicht (1, 1) wie in den Beispieldatendateien.

1.2.2 Umwandlung in nicht-gewichteten Digraphen mit Fakebatterien

Der gewichtete Digraph ist noch nicht für die Suche eines Wegs für den Roboter geeignet. Der Graph berücksichtigt nicht, dass eine Batterie abgelegt und später wieder aufgenommen werden kann. Zudem müssen für die Suche des Wegs eigentlich keine Distanzen zwischen den Knoten berücksichtigt werden, sondern es geht allein darum, welche Auswirkungen die Bewegung von einem Batteriefeld zu einem anderen auf den Zustand des Spielfelds hat.

Für jedes Batteriefeld gibt es eine endliche Zahl Ladungen, die eine auf dem Batteriefeld liegende Batterie haben kann. Der gewichtete Digraph wird in einen Graphen umgewandelt, der mehrere Knoten für jedes Batteriefeld besitzt. Jeder dieser Knoten steht für eine Batterieladung, die auf diesem Batteriefeld möglich ist. Zum Vergleich mit Abb. 8 zeigt Abb. 9 den Graphen nach der Umwandlung.[4]

[4] Für mehr Übersichtlichkeit wurde für den in Abb. 9 dargestellten Graphen darauf verzichtet, Schleifen im alten Graphen zu berücksichtigen. Mit Berücksichtigung der Schleifen gibt es im Feld 0,1 beispielsweise eine Kante vom Knoten 2,0 zu 0,-1. Zudem gäbe es auf Feld 4,3 auch die Ladungen 4 und 2, die durch Bewegung mit der Batterie mit Ladung 6 entstehen könnten. Auswirkungen der Schleifen werden später dargestellt.

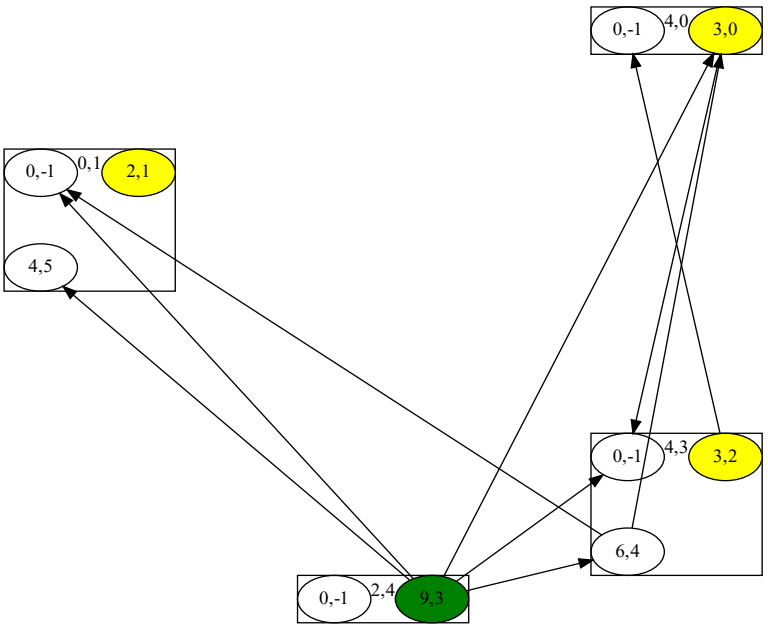


Abb. 9. Jeder der Kästen steht für ein Batteriefeld. Jeder Knoten ist mit „*Batterieladung*,*Batterie-ID*“ beschriftet. Die Batterie-ID wird von der Implementierung verwendet, um zu speichern, welche Batterieladung sich gerade auf einem Feld befindet (Batterien mit Ladung 0 haben keine ID (-1)). Knoten, welche die anfängliche Batterieladung der Ersatzbatterien repräsentieren, sind gelb ausgefüllt.

Eine Kante des Graphen stellt einen Weg dar, den der Roboter nehmen kann, wenn er eine Batterie mit bestimmter Ladung – dargestellt durch den Startknoten der Kante – von einem Batteriefeld aufgenommen hat. Der Endknoten einer Kante stellt die Batterieladung dar, mit der der Roboter nach der Bewegung eine Batterie auf einem Batteriefeld ablegt. Mit der anfänglichen Bordbatterie kann der Roboter also beispielsweise eine Batterie mit Ladung 3 auf das Feld 4,0 oder eine Batterie mit Ladung 6 auf das Feld 4,3 legen.

Batterien, die nicht die anfängliche Bordbatterie und nicht die anfänglichen Ersatzbatterien repräsentieren, heißen *Fakebatterien*. Es handelt sich nicht um völlig neue Batterien, sondern sie „entstehen“ durch die Bewegung des Roboters mit einer anderen Batterie. Es kommt vor, dass Knoten – wie die Batterie mit Ladung 3 auf Feld 4,0 – sowohl eine Ersatzbatterie als auch eine Fakebatterie darstellen. Im Folgenden wird der Begriff *Batterie* zusammenfassend für anfängliche Bordbatterie, anfängliche Ersatzbatterien und Fakebatterien genutzt, also allgemein für einen Knoten im Graphen.

Jedes Batteriefeld speichert den Knoten, der die Ladung derjenigen Batterie darstellt, die sich gerade auf dem Feld befindet. Diese Batterie heißt *aktive Batterie* des Batteriefelds. Zu Beginn stellen die gelb und grün (anfängliche Bordbatterie) ausgefüllten Knoten die aktiven Batterien dar. Bewegt sich der Roboter eine Kante entlang auf ein Batteriefeld, ist durch die Kante bekannt, welche Ladung die Batterie hat, die der Roboter auf dem Batteriefeld ablegt. Diese Batterie wird neue aktive Batterie des Batteriefelds. Der Roboter bewegt sich danach mit den Kanten der Batterie weiter, die eben noch aktiv war. Am Ende muss auf jedem Batteriefeld die Batterie mit Ladung 0 aktiv sein.

(Fake-)Batterien und zugehörige Kanten werden rekursiv ermittelt. Dazu werden alle Batteriefelder durchgegangen und jeweils die anfängliche Ersatzbatterie betrachtet.[5] Die Kanten für eine Batterie basieren auf den Kanten des zugehörigen Batteriefelds im alten Graphen. Alle vom Batteriefeld ausgehenden Kanten im alten Graphen, deren Distanz kleiner oder gleich der Ladung der betrachteten Batterie ist, kommen infrage. Ist die Parität der Distanz einer infrage kommenden Kante gleich der Batterieladung, gibt es auf jeden Fall eine Verbindung zu einer Batterie mit Ladung 0 auf dem Batteriefeld, auf das die Kante zeigt (Dazu muss es gegebenenfalls möglich sein, Ladung verfallen zu lassen. Dies wird mit dem von der Kante gespeicherten längeren Weg überprüft). Sollte ein entsprechender Knoten

[5] Es ist möglich, dass für Ersatzbatterien bereits Verbindungen berechnet wurden, wenn der Knoten gleichzeitig eine Ersatzbatterie und eine Fakebatterie, deren Verbindungen rekursiv durch Betrachtung einer anderen Ersatzbatterie bereits erstellt wurden, ist. In diesem Fall muss die Ersatzbatterie nicht betrachtet werden.

noch nicht existieren, wird er dem Graphen hinzugefügt. Ist die Distanz ungleich der Batterieladung, wird außerdem eine Verbindung zu einer Batterie auf dem anderen Batteriefeld hergestellt, die die Ladung *Distanz − Batterieladung* hat. Hat das Batteriefeld noch keinen entsprechenden Knoten, werden die Verbindungen dieser Batterie rekursiv nach demselben Schema erstellt.[6]

Einen Sonderfall nicht berücksichtigt, sollte der Roboter Batterieladung nur verfallen lassen, um eine Batterie leer werden zu lassen. In Abb. 10 sind mehr Batterien als nötig eingezeichnet:

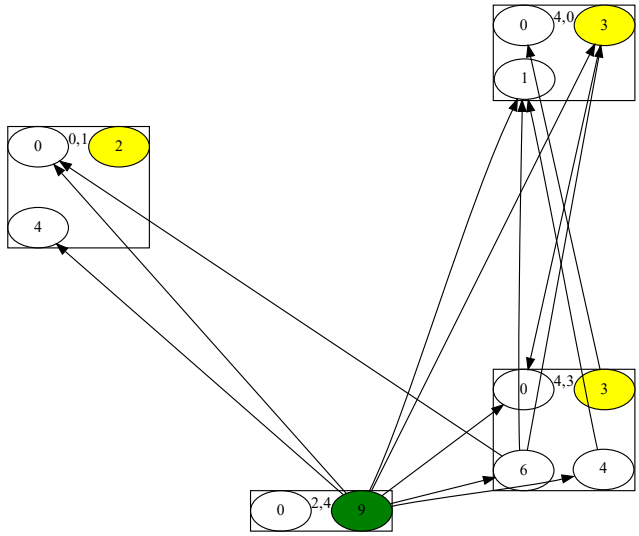


Abb. 10. Theoretisch sind mehr Knoten als die in Abb. 9 gezeigten möglich. In dieser Abbildung sind ein paar Beispiele dafür dargestellt. Mit der anfänglichen Bordbatterie kann auch eine Batterie mit Ladung 4 auf das Feld 4,3 gelegt werden oder eine Batterie mit Ladung 1 auf das Feld 4,0. (Batterie-IDs für mehr Übersichtlichkeit entfernt.)

Es ist jedoch mit der Bordbatterie weder sinnvoll, eine Batterie mit Ladung 4 auf das Feld 4,3 zu legen, noch eine Batterie mit Ladung 1 auf das Feld 4,0. Mit der 4er-Batterie auf Feld 4,3 sind alle Wege möglich, die auch mit der 6er-Batterie möglich wären – mit der 6er-Batterie sind aber noch mehr Wege möglich, weil die Ladung zu Beginn nicht unnötig verringert wird, wie bei der 4er-Batterie. Die 1er-Batterie auf Feld 4,0 ist aufgrund der gleichen Parität gleichbedeutend mit der 3er-Batterie, hat aber wiederum weniger mögliche Wege.[7] Der Roboter lässt daher nur Ladung verfallen, wenn die Batterie bei Ankunft auf dem Zielfeld leer ist.

Sonderfall 1 – Batterieladung verfallen lassen, ohne dass Batterie bei Ankunft leer ist

Es gibt einen Sonderfall, bei dem es nötig ist, Batterieladung schon vom ersten zum zweiten Feld und nicht erst vom zweiten zum dritten Feld verfallen zu lassen. Dies ist der Fall, wenn es auf dem Weg vom zweiten zum dritten Feld nicht möglich ist, Ladung verfallen zu lassen:

[6] Sollte die neue Batterie keine Verbindungen zu anderen Batterien haben, gibt es trotzdem eine Verbindung im Graphen und der Knoten wird hinzugefügt, allerdings hat der Roboter nur die Möglichkeit, die Ladung der neuen Batterie auf 0 zu reduzieren, wenn er diese als letztes besucht und die zusammenhängenden freien Felder des Batteriefelds es zulassen.

[7] Mit Berücksichtigung der Schleifen gäbe es tatsächlich beide Batterien, allerdings nicht mit den dargestellten Kanten.

	1		
0	**2**	**0**	
	6		

Abb. 11. Eine Lösung für diese Spielsituation sieht so aus: Bewege dich mit der Bordbatterie so zum Feld mit der Ersatzbatterie mit Ladung 2, dass die Bordbatterie auf diesem Feld mit verbleibender Ladung 1 abgelegt wird. Gehe mit der 2er-Batterie ein Feld nach unten, dann zwei Felder nach oben und eines nach rechts oder links (und tausche währenddessen Batterien entsprechend).

Um diesen Fall zu erkennen, speichert im alten Graphen jedes Batteriefeld, Verbindungen welcher Länge von diesem Batteriefeld wegführen, auf denen es nicht möglich ist, beliebige Batterieladungen verfallen zu lassen. Der Roboter kann beliebige Ladung verfallen lassen, wenn der längere Weg die Länge 3 (kürzerer Weg hat Länge 1) bzw. 4 (kürzerer Weg hat Länge 2) hat.[8] Für das Batteriefeld, auf dem sich zu Beginn die 2er-Ersatzbatterie befindet, wird eine Länge gespeichert, nämlich 1. Immer, wenn Wege zum Batteriefeld mit der 2er-Ersatzbatterie hinzugefügt werden, wird nach Möglichkeit[9] auch eine Kante hinzugefügt, bei der eine Batterie mit Ladung 1 auf dem Feld abgelegt wird. Die Graphen zu obigem Spielfeld zeigt Abb. 12.

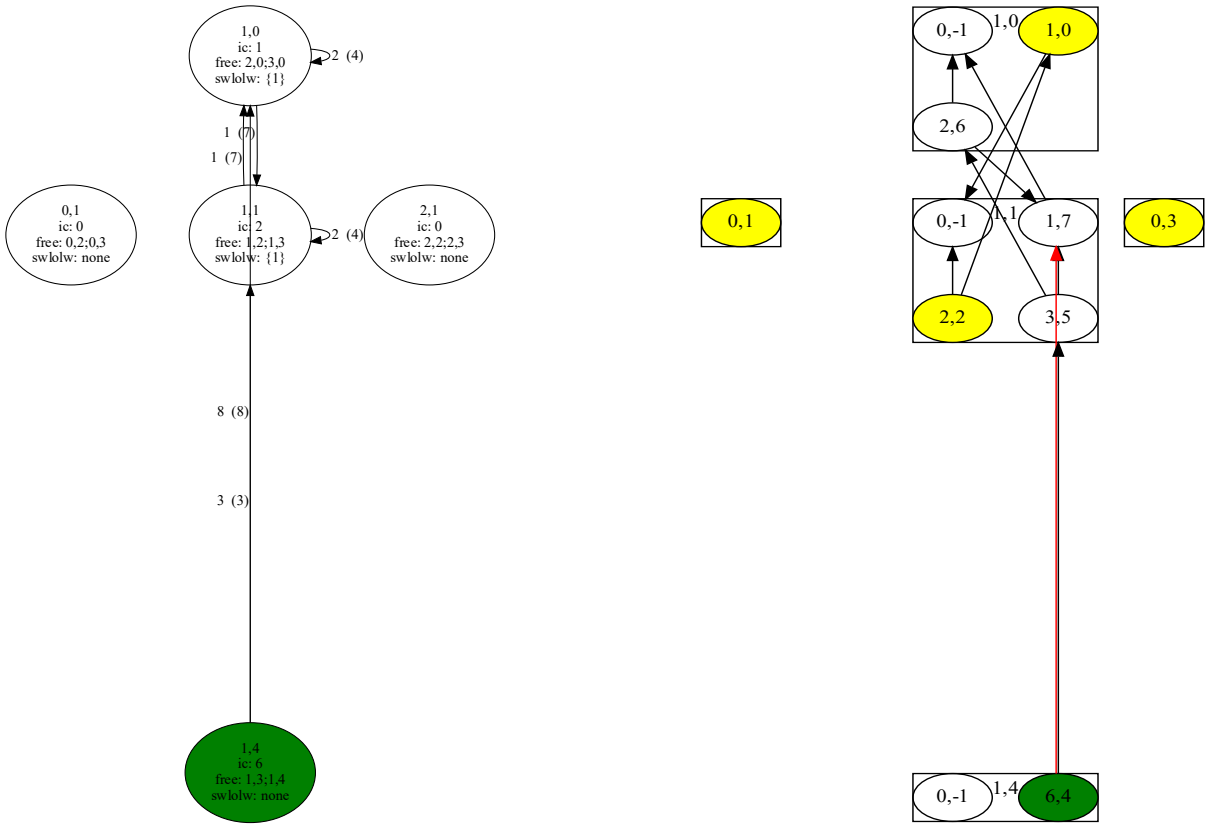


Abb. 12. Linker Graph: Batteriefelder. Die zusätzliche Angabe „swlolw“ steht für „small way lengths without longer way“. Die rotgefärbte Kante im rechten Graphen wird hinzugefügt, weil der Roboter auf dem Weg von Feld 1,1 zu Feld 1,0 nicht beliebige (gerade) Ladung verfallen lassen kann.

[8] Man könnte meinen, dass es ebenfalls nötig ist, als Längen 1 und 2 abzuspeichern, wenn das Batteriefeld nur 0 oder 1 zusammenhängendes freies Feld hat. Hat ein Batteriefeld allerdings weniger als 2 zusammenhängende freie Felder, dann gibt es einen von diesem Batteriefeld wegführenden Weg, der 1 oder 2 Felder lang ist und zu dem es keinen längeren Weg gibt. Für diesen Weg wird daher ohnehin eine solche Länge gespeichert. Daher muss dieser Fall nicht berücksichtigt werden.

[9] Nicht möglich ist dies, wenn der Roboter auf dem Weg zum Batteriefeld mit der 2er-Ersatzbatterie keine entsprechende Ladung verfallen lassen kann.

Sonderfall 2 – Schleifen

Mit einer neu aufgenommenen Batterie kann der Roboter auch direkt zu dem Batteriefeld zurückkehren, von dem er die Batterie aufgenommen hatte. Mit den Schleifen im alten Graphen wird so auch die folgende Spielsituation gelöst:

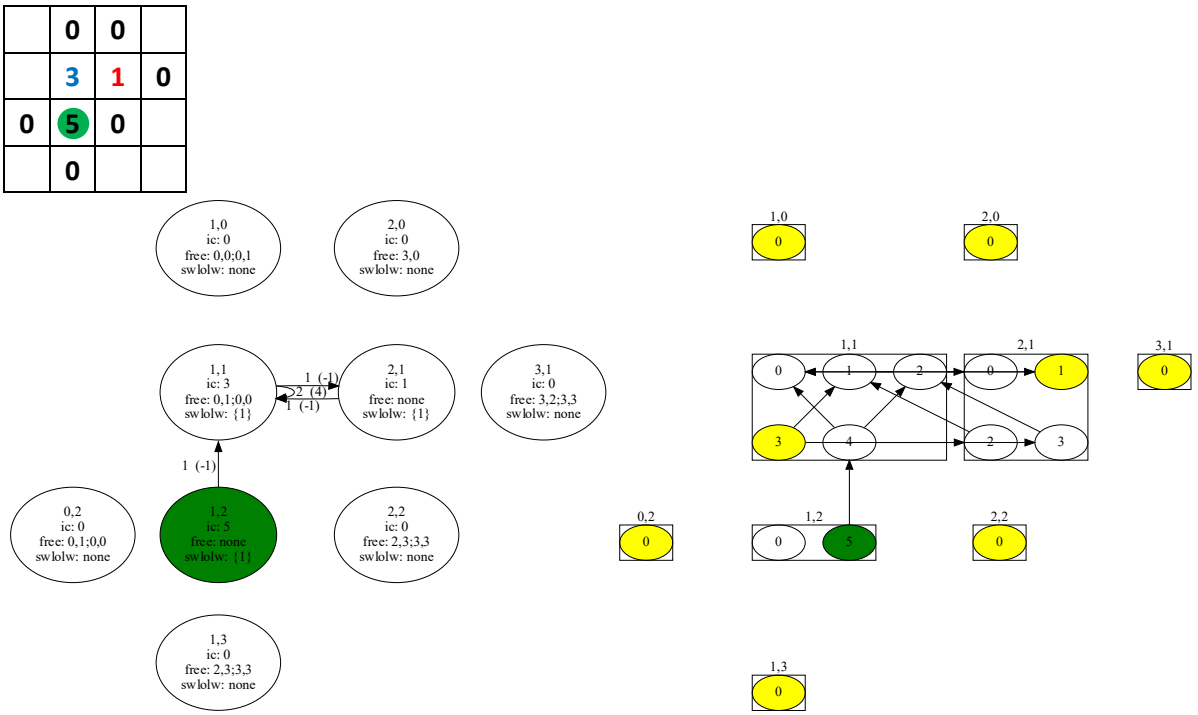


Abb. 13. Diese Spielsituation kann nur gelöst werden, indem Batterien auf das Feld, von dem sie aufgenommen wurden, wieder zurückgelegt werden. Eine mögliche Bewegungsabfolge des Roboters ist `olrlourrl` (o: oben, r: rechts, u: unten, l: links; Schritte mit anfänglicher Bordbatterie: grün, 3er-Ersatzbatterie: blau, 1er-Ersatzbatterie: rot).

Wie im Graphen rechts zu sehen, entstehen durch die Schleifen Kanten, die nur bewirken, dass die aktive Batterie des Batteriefelds durch eine andere ersetzt wird, weil der Roboter zu dem Feld zurückkehrt, von dem er eine Batterie aufgenommen hatte. Diese Kanten verlaufen innerhalb eines Kastens, der ein Batteriefeld darstellt.

1.3 Hamiltonpfad und Backtracking

In dem Graphen mit Fakebatterien wird ein Weg gesucht, der auf jedem Batteriefeld den Knoten besucht, der die Ladung 0 darstellt.[10] Dieses Problem weist starke Ähnlichkeiten mit dem Hamiltonpfadproblem auf. Aufgrund der Ähnlichkeit ist davon auszugehen, dass dieses Problem wie das Hamiltonpfadproblem NP-vollständig ist. Im Unterschied zum Hamiltonpfad verändert sich der Graph jedoch während der Suche, weil sich die aktive Batterie eines Batteriefelds ändert.

Grundlage für die Suche nach einem Weg bildet das im Folgenden dargestellte Backtrackingverfahren. Die Eingabe zu Beginn ist:

- *aktuelle Batterie*: Knoten, der die anfängliche Bordbatterie des Roboters darstellt
- *neue aktive Batterie*: Knoten mit Ladung 0 des Roboterstartfelds

1. Eingabe:
    - *aktuelle Batterie*: Diese Batterie nimmt der Roboter vom aktuellen Batteriefeld auf und bewegt sich mit ihr weiter.

[10] Batteriefelder, auf denen bereits zu Beginn eine leere Batterie liegt, werden nicht besucht.

- *neue (aktive) Batterie*: Die Batterie, die der Roboter auf dem Feld ablegt, von dem er die *aktuelle Batterie* genommen hat. *Aktuelle* und *neue Batterie* haben dasselbe Batteriefeld. Die *aktuelle Batterie* ist im Moment noch die aktive Batterie dieses Batteriefelds und wird durch die *neue aktive Batterie* ersetzt.

2. Setze die aktive Batterie des Batteriefelds, zu dem die beiden Batterien gehören, auf *neue Batterie*. Wenn die Ladung der *neuen Batterie* 0 ist: Speichere, dass dieses Batteriefeld abgeschlossen ist; es sollte nicht mehr besucht werden. Das Batteriefeld wird als inaktiv markiert. Ein Batteriefeld ist aktiv, wenn die aktive Batterie des Feldes geladen ist, ansonsten ist es inaktiv.
3. Falls alle Batteriefelder inaktiv sind:
    a. Überprüfe, ob es das Batteriefeld zulässt, dass die Ladung der *aktuellen Batterie* „verfahren“ wird. Ladung zu „verfahren“ ist dann möglich, wenn die Ladung der *aktuellen Batterie* 2 ist und das Batteriefeld mindestens ein Nachbarfeld hat; ist die Ladung größer, sind zwei zusammenhängende freie Felder nötig. Wenn ja: Gib den letzten Teilweg bestehend aus *aktueller Batterie* und *neuer Batterie* zurück.
4. Gehe alle Kanten durch, die von der *aktuellen Batterie* wegführen:
    a. Versuche, das Batteriefeld der Batterie, auf die die Kante zeigt, als nächstes zu besuchen: Dafür muss das Batteriefeld aktiv sein.
        i. Wenn das Batteriefeld besucht werden kann: Versuche, mit diesem Verfahren einen Weg zu finden, mit der Eingabe: *aktuelle Batterie*: Die aktive Batterie des Batteriefelds; *neue Batterie*: Die Batterie, auf die die Kante zeigt.
        ii. Falls ein Weg gefunden wurde: Gib diesen Weg zurück, füge allerdings den genutzten Teilweg bestehend aus *neuer Batterie* und *aktueller Batterie* zum Wegbeginn hinzu.
5. Es wurde kein Weg gefunden: Setze die aktive Batterie des Batteriefelds zurück auf *aktuelle Batterie*, markiere das Batteriefeld als nicht abgeschlossen (aktiv).
6. Gib zurück, dass kein Weg gefunden wurde.

Dieser Algorithmus gibt einen Weg zurück, der die besuchten Knoten darstellt. Indem jede Kante des *neuen* Graphen speichert, mit welcher Kante des alten Graphen sie „entstehen“ konnte, ist es auch möglich, den genauen Weg auszugeben. Dazu müssen die Kanten des *alten* Graphen jeweils die Wege (= Liste von Positionen) speichern, die zwischen den Batteriefeldern gefunden wurden (kürzester Weg und Weg länger als 2, wenn es einen solchen gibt).

1.3.1 Backtracking-Optimierung I: Memoisation

Obiger Algorithmus wird eventuell mehrfach für dasselbe Teilproblem aufgerufen. Zur Erkennung gleicher Teilprobleme reicht es nicht aus, nur die Eingabe des Algorithmus auf Gleichheit zu überprüfen, weil es vorkommen kann, dass dieselbe Batterie besucht wird, aber mit unterschiedlichen vorher genommenen Wegen. Daher müssen nicht nur aktuelle und neue Batterie, sondern auch die Menge aller aktiven Batterien auf den Batteriefeldern berücksichtigt werden. Dabei ist es irrelevant, in welcher Reihenfolge die Batteriefelder bisher besucht wurden, denn es geht allein darum, welche Batteriefelder mit welchen aktiven Batterien noch besucht werden müssen, und von welcher Batterie aus. Ein Cache speichert daher, mit welchen Eingaben für den Algorithmus und mit welchen Mengen aller aktiven Batterien kein Weg gefunden wurde. Der Cache speichert keine gefundenen Teilwege, weil nach dem Finden des letzten Teilwegs in Schritt 3a der gesamte Weg bekannt ist.

Ohne Memoisation müssten mit dem Backtrackingverfahren maximal $n!$ Wege getestet werden, wobei n für die Anzahl aller Knoten und nicht für die Anzahl der Batteriefelder steht. Teilmengen aus der Menge aller Knoten gibt es maximal 2^n: Für jeden Knoten gibt es zwei Zustände; er kann entweder der aktive Knoten des Batteriefelds sein oder nicht. Die tatsächliche Anzahl der möglichen Teilmengen ist

meist niedriger, weil jedes Batteriefeld nur eine aktive Batterie haben kann. Für die aktuelle Batterie gibt es n Möglichkeiten, weshalb sich maximal $n2^n$ Elemente im Cache befinden können. Durch Memoisation wird die Laufzeit des Verfahrens daher auf $O(n2^n)$ beschränkt.

Die verwendete Memoisation basiert auf dem von R. Bellman vorgestellten Verfahren [2] für das Problem des Handlungsreisenden (travelling salesperson problem), das Dynamische Programmierung nutzt. Das Problem des Handlungsreisenden ist allerdings mit dem Hamilton*kreis*problem verwandt, es wird also nach einer Route gesucht, die zum Ausgangspunkt zurückführt. Aus diesem Grund erreicht das Verfahren von Bellman eine Laufzeit von $O(n^2 2^n)$, weil – übertragen auf das „Batteriefeldproblem" – nicht nur die aktuelle Batterie im Cache gespeichert werden muss, sondern auch die Zielbatterie.

1.3.2 Backtracking-Optimierung II: Zusammenhang des Graphen überprüfen

Gerade bei größeren Spielfeldern mit zahlreichen Batterien kann es vorkommen, dass der Algorithmus einen Weg sucht, obwohl es für die aktuelle Spielsituation keine Lösung gibt. Um dies zu verhindern, ist es sinnvoll, bei jedem Aufruf zu prüfen, ob von der aktuellen Batterie aus alle anderen Batteriefelder erreichbar sind.[11] Ist das nicht der Fall, kann direkt zurückgegeben werden, dass kein Weg existiert. Begonnen wird beim Batteriefeld, auf dem sich die aktuelle Batterie befindet. Jedes Batteriefeld speichert, welche anderen Batteriefelder von diesem Batteriefeld aus erreichbar sind. Dies unterscheidet sich vom Batteriefeldgraphen, weil der alte Graph Kanten zwischen allen Batteriefeldern besitzt, sofern es einen Weg zwischen ihnen gibt. Vom zweiten Graphen wird aber nicht jeder dieser Wege tatsächlich genutzt. Beim Prüfen, ob der Graph zusammenhängend ist, wird also nicht beachtet, welche Batterien gerade aktiv sind, sondern nur, welche Batteriefelder theoretisch, mit welchen aktiven Batterien auch immer, erreicht werden könnten.

Um zu prüfen, ob der Graph zusammenhängend ist, wird die Anzahl der aktiven Batteriefelder gespeichert. Durch Breitensuche (Tiefensuche ist auch möglich) werden alle Nachbarfelder des Startfelds besucht, danach alle noch unbesuchten Nachbarfelder der Nachbarfelder und so weiter. Für jedes besuchte Feld wird 1 von der Anzahl der aktiven Batteriefelder abgezogen. Sobald diese Anzahl 0 ist, wurde jedes Batteriefeld einmal besucht und es ist folglich möglich, alle verbleibenden Batteriefelder von der aktuellen Batterie aus zu erreichen. Wurden alle Felder besucht, für die dies möglich ist, aber die Anzahl ist nicht 0, können nicht alle noch aktiven Batteriefelder erreicht werden.

Dass mit dem Batteriefeld, auf dem sich die aktuelle Batterie befindet, begonnen wird, bedeutet nicht, dass dieses Batteriefeld sofort als besucht gezählt wird. Hatte die neue Batterie für dieses Batteriefeld nicht die Ladung 0, muss das Batteriefeld durch die Breitensuche erneut besucht werden.

1.3.3 Backtracking-Optimierung III: Sortierung der Kanten nach Relevanz

Mit einer Heuristik lässt sich die Laufzeit des Verfahrens in vielen Fällen verbessern. In Schritt 4 des oben dargestellten Algorithmus werden alle Kanten zu anderen Batterien durchgegangen. Werden die Kanten zuerst getestet, für die es am wahrscheinlichsten ist, dass sie auf dem richtigen Weg liegen, findet der Algorithmus schneller eine Lösung. Eine Sortierung der Kanten wird an zwei Stellen vorgenommen: Direkt nachdem der zweite Graph erstellt wurde und vor Schritt 4 in obigem Algorithmus.

Die Sortierung nach der Erstellung des zweiten Graphen sortiert die Kanten nach der Batterieladung, die die Batterie hat, auf die die Kante zeigt. Je nach Spielsituation kann es sinnvoll sein, entweder die Kante mit der größten oder die mit der kleinsten zuerst zu betrachten. Wege werden für die Beispieldaten deutlich schneller gefunden, wenn zuerst die Kanten betrachtet werden, die auf Batterien mit

[11] Es wird also nicht „Zusammenhang" im eigentlichen Sinne geprüft. Es wird nicht geprüft, ob alle Knoten (= alle Batterien, d.h. anfängliche Bordbatterie, Ersatzbatterien und Fakebatterien) erreicht werden können, denn zum Finden des Weges müssen alle Batteriefelder, nicht alle Knoten besucht werden.

der größten Ladung zeigen. Ist für eine Spielsituation bereits bekannt, dass der Roboter in der Lösung keine Ladung verfallen lässt, ist es sinnvoll, dieses Wissen zur Optimierung des Algorithmus zu nutzen und zuerst Batterien mit geringerer Ladung zu besuchen. Es ist außerdem möglich, die Kanten so zu sortieren, dass zuerst Kanten betrachtet werden, die nicht auf eine Batterie im eigenen Feld zeigen. Dadurch wird einer höheren Laufzeit entgegengewirkt, die durch die Berücksichtigung von meistens irrelevanten Schleifen entsteht. Ebenso ist es sinnvoll, Kanten zuletzt zu betrachten, die zu Batterien ohne wegführende Kanten führen.

Direkt vor Schritt 4 können die bereits sortierten Kanten in zwei Kategorien eingeteilt werden: Die Kanten, die zu einer Batterie führen, die auf einem noch nicht besuchten Batteriefeld liegen, und die, die zu einem bereits besuchten Batteriefeld führen. Insbesondere für `beispieldaten2.txt` ist es wichtig Kanten, die zu noch nicht besuchten Batteriefeldern führen, zuerst zu betrachten. Der von der Implementierung gefundene Weg für `beispieldaten2.txt` besucht zunächst nahezu alle Batteriefelder einmal. Danach befindet sich auf (fast) jedem Feld eine Batterie mit Ladung 1 und durch Besuchen dieser übrigen Felder werden alle Batterien leer. Ohne die Optimierung würde der Algorithmus für `beispieldaten2.txt` viele Wege ausprobieren, die zu keiner Lösung führen.

1.4 Erweiterung Aufgabenteil a): Teleportationsfelder

Auf dem Spielbrett können sich Teleportationsfelder befinden. Teleportationsfelder treten immer paarweise auf. Bewegt sich der Roboter auf ein Teleportationsfeld, wird er auf das andere der beiden Teleportationsfelder gesetzt. Dabei verringert sich die Ladung der Bordbatterie um 1. Der Roboter darf sich nach einer Teleportation direkt wieder zurückteleportieren lassen, ohne zuerst ein anderes Feld zu betreten.

Ziel der Umsetzung dieser Erweiterung ist, nur die Erstellung des ersten Graphen anzupassen. Durch die Teleportationsfelder werden zwischen den Batteriefeldern mehr Kanten möglich. In Abb. 14 gibt es vom Feld 3,1 nicht nur eine Kante mit Distanz 8 zum Feld 5,7, sondern zum Beispiel auch eine Kante mit Distanz 5. Für die Kante mit Distanz 5 wird der Roboter vom Feld 4,1 zum Feld 4,4 teleportiert, dann bewegt er sich zu Feld 5,4 und wird zu Feld 4,7 teleportiert.

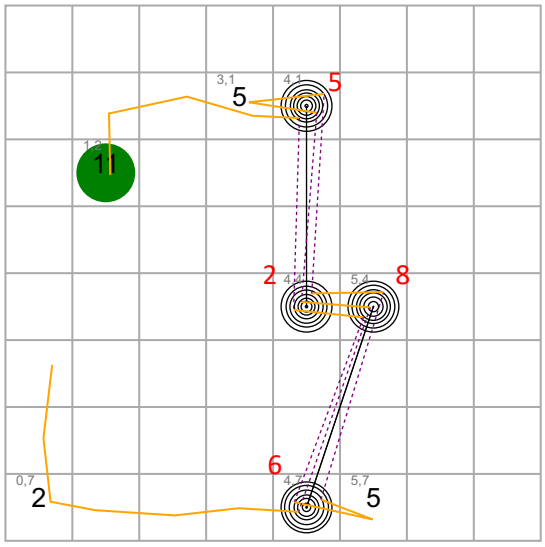


Abb. 14. Ein Spielbrett mit Teleportationsfeldern, mit eingezeichneter Lösung (gestrichelte Linien zeigen Teleportation an). Ein Teleportationsfeldpaar ist durch eine schwarze Linie verbunden. In rot sind die initialisierten Distanzen eingezeichnet (Abschnitt 1.4.1). Die Spielsituation ist ähnlich wie stromrallye0.txt aufgebaut.

Um diese zusätzlichen Kanten zu bestimmen, werden die Teleportationsfelder zunächst als Batteriefelder betrachtet und es werden Kanten zwischen den Batterie- und Teleportationsfeldern per Maze-Routing erstellt. Nach der nun folgenden Umwandlung des Graphen gibt es nur noch Knoten, die Batteriefelder darstellen. Für jedes Batteriefeld werden Wege zu allen anderen Batteriefeldern gesucht, wobei die Wege Teleportationsfelder nutzen. Mit den Distanzen dieser Wege werden Kanten zwischen den Batteriefeldern gebildet. Es ist es möglich, dass es zwischen Batterien dadurch Wege mit Distanzen beider Paritäten gibt. Zwischen den Batteriefeldern gibt es unter Umständen Kanten, die überflüssig sind, da es andere Kanten mit derselben Parität, aber niedrigerer oder gleicher Distanz gibt. In Abb. 14 gibt es zwischen den Feldern 3,1 und 5,7 auch Distanzen länger als 5. Werden die Teleportationsfelder in unterschiedlicher Reihenfolge besucht, sind beispielsweise auch die Distanzen 9 (3,1; 4,1; 4,4; bewegen zu 4,7; 5,4; bewegen zu 5,7), 6 (3,1; 4,1; 4,4; bewegen zu 5,7) und 7 (3,1; bewegen zu 5,4; 4,7;

5,7) möglich. Von diesen Distanzen sind jedoch nur zwei relevant, nämlich für jede Parität die kleinste Distanz. Von Feld 3,1 zu Feld 5,7 gibt es daher im resultierenden Graphen nur die Distanzen 5 und 6 (die Distanz 8, die ohne Teleportationsfelder erreicht wird, ist größer als 6 und ist daher irrelevant):

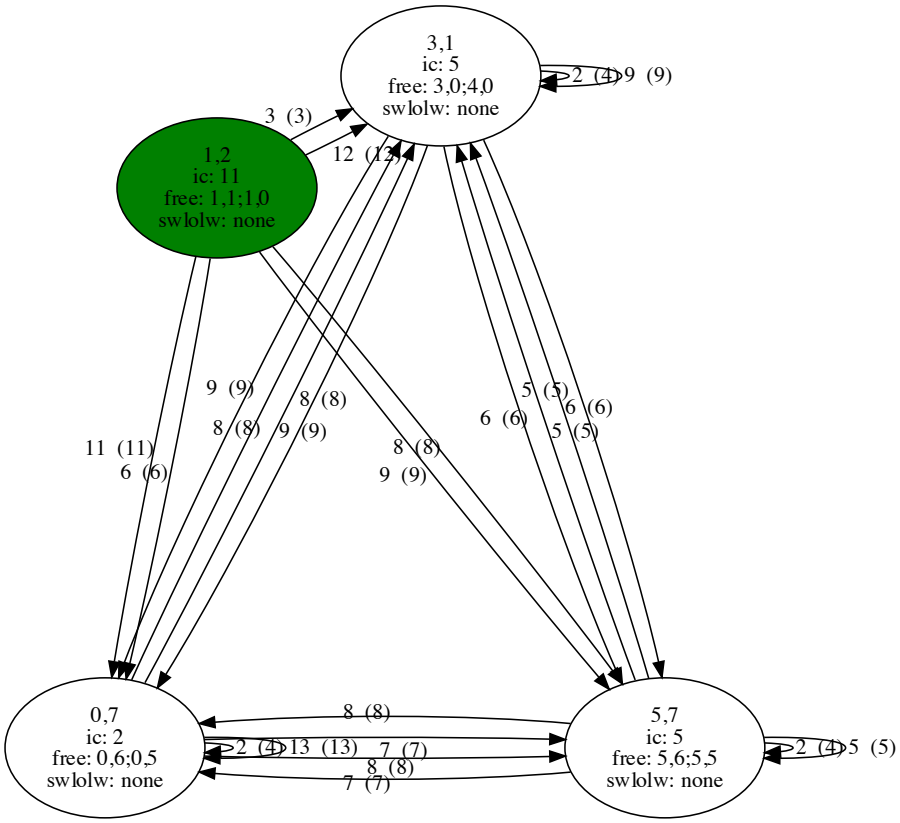


Abb. 15. Dieser Graph entsteht für die Spielsituation aus Abb. 16, nachdem alle Wege, die durch Teleportationsfelder möglich sind, zu Kanten zwischen Batteriefeldern „konvertiert" wurden.

Statt die Teleportationsfelder als Batteriefelder zu betrachten, wäre es auch möglich, den verwendeten Lee-Algorithmus zu erweitern: Wird ein Teleportationsfeld besucht, werden nicht alle Nachbarn des Teleportationsfeldes mit einer Distanz markiert, sondern alle Nachbarn des zugehörigen Teleportationsfeldes. Damit aber verschiedene Wege, mit und ohne Teleportationsfelder und mit verschiedenen Paritäten, gefunden werden, müssten die Felder mit verschiedenen Distanzen markiert werden. Zudem müsste nun für jeden Weg zwischen zwei Batteriefeldern Maze-Routing angewendet werden, selbst wenn ein Weg durch die Optimierung auf einfache Weise gefunden wurde. In vielen Fällen effizienter ist das verwendete Verfahren, das auf dem im folgenden Abschnitt vorgestellten Algorithmus basiert.

1.4.1 Dijkstra-ähnlicher[12] Algorithmus zur Suche aller kürzesten Wege mit Teleportation

Ein Weg, der durch Teleportationsfelder entsteht, beginnt bei einem Batteriefeld, auf dem Weg wird der Roboter mindestens einmal teleportiert, und endet auf einem Batteriefeld. Das Zielfeld kann auch identisch mit dem Startfeld sein – dann entstehen Schleifen wie in Abb. 17 mit ungerader Distanz. Mit einem Algorithmus sollen die kürzesten Wege von einem Startbatteriefeld aus zu allen Batteriefeldern (inklusive Startbatteriefeld) bestimmt werden. Zu jedem Batteriefeld gibt es zwei kürzeste Wege: einen mit gerader und einen mit ungerader Distanz. Dies wird durch einen Algorithmus erreicht, der einem Dijkstra-Algorithmus ohne Zielfeld ähnlich ist. Es werden solange Felder besucht, bis keine Felder mehr besucht werden können. Wird vom Algorithmus ein Batteriefeld besucht, dann ist der zu diesem Batteriefeld gefundene Weg ein kürzester. Von einem Batteriefeld aus findet keine Relaxation statt. Wird ein Teleportationsfeld besucht, werden die Distanzen zu den noch nicht besuchten Nachbarfeldern

[12] Der Algorithmus ist dem Dijkstra-*Algorithmus* ähnlich, nicht Edsger W. Dijkstra.

wie beim Dijkstra-Algorithmus üblich aktualisiert. Jedes Teleportationsfeld hat dabei auch eine Kante zu dem mit dem Teleportationsfeld verbundenen Feld. Diese Kante hat die Distanz 1.[13]

Zu jedem Feld muss es zwei Distanzen geben: Eine Distanz ist gerade, eine ungerade. Dies bedeutet, dass der Algorithmus jeden Knoten zweimal besuchen muss (natürlich unter der Voraussetzung, dass es eine Distanz mit beiden Paritäten gibt). Dies wird gelöst, indem jeder Knoten durch zwei Knoten dargestellt wird, einen mit gerader und einen mit ungerader Distanz.

Der Roboter kann auf einem Teleportationsfeld nicht stehenbleiben, wenn er es direkt betritt, sondern er muss hierfür zuerst das verbundene Teleportationsfeld betreten. Deshalb wird anders als beim Dijkstra-Algorithmus üblich zu Beginn das Startfeld nicht mit Distanz 0 initialisiert. Stattdessen wird für jedes Teleportationsfeld als Initialwert die Distanz zum verbundenen Teleportationsfeld erhöht um 1 verwendet – sofern das verbundene Teleportationsfeld vom Startfeld aus erreichbar ist. In Abb. 14 sind die Distanzen vom exemplarisch ausgewählten Startfeld 3,1 zu den Teleportationsfeldern in rot eingezeichnet. Beispielsweise hat der Weg ohne Teleportation zum Feld 4,4 die Distanz 4, weshalb Feld 4,1 als initiale Distanz 4+1=5 (+1 für Teleportation) erhält. Diese Besonderheit wird auch bei der Relaxation berücksichtigt.

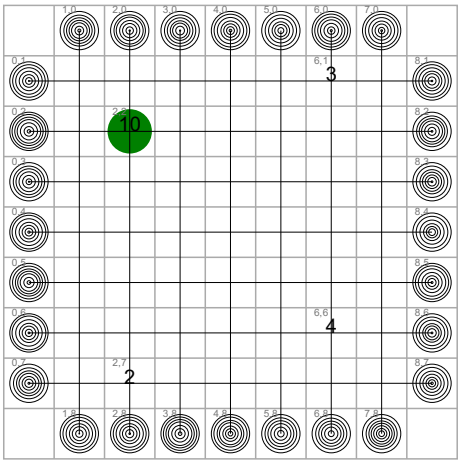
Nachdem die neuen Kanten hinzugefügt wurden, werden die Kanten „aufgeräumt“. Unter den Kanten, deren Wege Teleportationsfelder nutzen, kann es keine überflüssigen Kanten geben. Es kann aber sein, dass eine Kante, deren Weg Teleportationsfelder nutzt, eine Kante, deren Weg keine Teleportationsfelder nutzt, überflüssig macht – oder umgekehrt. Eine Kante ist genau dann überflüssig, wenn es eine andere Kante gibt, deren Distanz dieselbe Parität hat und deren Distanzen (die normale Distanz und die eines eventuellen längeren Weges) kleiner oder gleich sind. Alle diese Kanten werden entfernt.

1.4.2 Neue Möglichkeiten für zusammenhängende freie Felder

Auf einem Weg mit Teleportationsfeldern kann der Roboter immer Ladung verfallen lassen. Wird der Roboter auf einem Weg teleportiert, ist der Weg mindestens 3 Felder lang und der Roboter kann sich nach dem Teleportieren entscheiden, sich zurückteleportieren zu lassen. Dies bedeutet aber auch, dass für die zusammenhängenden freien Felder neue Möglichkeiten entstehen. Befindet sich ein Teleportationsfeld direkt neben einem Batteriefeld, kann dieses mit einer beliebigen Ladung zuletzt besucht werden. Gleiches gilt, wenn sich zwischen dem Teleportationsfeld und dem Batteriefeld ein freies Feld befindet.

1.4.3 Roboter kann sich über Spielfeldgrenzen hinwegbewegen

Durch Platzieren von Teleportationsfelder am Rand ist eine weitere interessante Erweiterung möglich: Der Roboter kann sich nun über Spielfeldgrenzen hinwegbewegen und betritt das Spielbrett auf der anderen Seite wieder.[14]



[13] Zwischen zwei Teleportationsfeldern, die zu demselben Teleportationsfeldpaar gehören, gibt es zwei Kanten: Eine mit der Distanz 1 und eine, die per Maze-Routing bestimmt wurde. Letztere Kante ist ebenfalls relevant, weil es sein kann, dass diese Kante eine gerade Distanz hat (im Gegensatz zur ungeraden Distanz bei der Teleportation).

[14] Auf diese Weise reduziert sich die Ladung der Bordbatterie beim „Seitenwechsel“ um 1. Dies kann verhindert werden, indem die Ladung der Bordbatterie beim Teleportieren nicht verändert wird, also die entsprechenden Kanten die Distanz 0 erhalten.

1.5 Spielfeldgenerierung

Folgende Kriterien, Gegebenheiten und Aspekte sind für die Spielfeldgenerierung von besonderer Bedeutung.

1. Mit Kenntnis des Algorithmus sollte es nicht möglich sein, einen Großteil der Lösung „abzulesen". Dazu zählt auch, dass die generierte Spielsituation möglichst wenig „Hinweise" gibt, also sollten beispielsweise Felder, die mehrfach besucht werden, durchschnittlich keine höhere Ladung als andere haben.
2. Eine Unterscheidung zwischen „aufwendig" zu lösenden und „schwierig" zu lösenden Spielsituationen ist sinnvoll, aber nicht immer möglich. Aufwendig ist eine Spielsituation vor allem dann, wenn viele Wege für den Roboter möglich erscheinen. Dies bewirkt, dass zum Lösen der Spielsituation viel Zeit aufgewendet werden muss, obwohl die Spielsituation nicht unbedingt „schwierig" ist. Schwierig wird die Spielsituation dann, wenn die menschliche Spielerin/der menschliche Spieler viel Wissen über Lösungsstrategien benötigt und auch „Sonderfälle" erkennen können muss (sozusagen „Intelligenz" statt „Rechenleistung").
3. Um variierende Spielsituationen zu erhalten, wird ein (deterministischer) Zufallszahlengenerator verwendet. Mit gleichem Startwert für den PRNG und gleichen Parametern sollte das gleiche Spielfeld generiert werden; die Generierung darf insbesondere nicht vom aktuellen Arbeitsspeicher oder variierenden Hashwerten (durch die z.B. die Elemente einer Menge, eines Binärbaums oder ähnlichem anders sortiert werden) abhängen.[15]
4. Die Schwierigkeit sollte möglichst stark von Parametern und möglichst wenig vom Startwert für den Zufallszahlengenerator abhängen.
5. Besonders gut ist die Generierung, wenn mit entsprechendem Startwert und entsprechenden Parametern theoretisch alle möglichen Spielsituationen erzeugt werden können. Dies bedeutet jedoch nicht, dass alle Spielsituationen mit der gleichen Wahrscheinlichkeit generiert werden.
6. Es sollte möglichst wenige unterschiedliche Abfolgen von besuchten Batteriefeldern geben, die eine Lösung darstellen („nur eine Lösung" wäre ungenau, da der Roboter zwischen zwei Batteriefeldern meist aus verschiedenen Wegen wählen kann, bei der Lösung kommt es jedoch nur auf die Abfolge der Batteriefelder an). Sind viele Lösungen möglich, ist auch die Spielsituation einfacher zu lösen. Zu verhindern, dass es mehrere mögliche Batteriefeldabfolgen gibt, ist sehr aufwändig – zumindest mit dem zur Generierung verwendeten Verfahren.
7. Im Einklang mit den Beispieldaten werden keine Batterien mit Ladung 0 generiert. Batterien mit Ladung 0 sind, wie in den Darstellungen einiger Sonderfälle oben, gut dazu geeignet, besondere Spielsituationen darzustellen. Eine Spielsituation ist jedoch grundsätzlich schwerer, wenn alle Batterien zu Anfang geladen sind, weil dadurch mehr Wege möglich werden.
8. Zur Lösung einer Spielsituation wird, wie schon in Punkt 2 angedeutet, grundsätzlich mehr Zeit benötigt, wenn es mehr Batterien gibt. Eine Spielsituation wird tendenziell schwieriger, wenn es viele Teilwege gibt, die so scheinen, als würden sie zur Lösung gehören. Mehr solcher „Irrwege" gibt es, wenn es im zweiten Graphen mehr Kanten gibt. Und mehr Kanten im zweiten Graphen entstehen dadurch, dass sich mit der (anfänglichen) Batterie eines Batteriefelds viele andere Batterien erreichen lassen. Wenn die Batterien gut verbunden sind, wird es jedoch wahrscheinlicher, dass mehrere Batteriefeldabfolgen eine Lösung darstellen.
9. Es werden keine „Muster" generiert, wie sie zum Beispiel in `beispieldaten5.txt` auftreten. In `beispieldaten5.txt` gibt es am linken Rand eine Aneinanderreihung von Batterien der Ladung 1 und oben rechts ein Muster mit Batterien der Ladung 2 (siehe Abb. 16). Muster bewirken nicht, dass die Spielsituation schwerer wird. Im Gegenteil sorgen sie dafür, dass es mehr Batterien gibt,

[15] Beispielsweise randomisiert Python standardmäßig Hashwerte pro Python-Prozess, um Denial-of-Service-Attacken zu verhindern (siehe docs.python.org/3/reference/datamodel.html#object.__hash__).

die Spielerin oder der Spieler es aber leichter hat, weil von dem Muster nur ein Teil gelöst werden muss. Ließe man in Abb. 16 die blauen Batterien weg, wäre das Spielfeld genauso schwer zu lösen, da sich der für die roten Batterien gefundene Weg auf die blauen Batterien übertragen lässt.

		2
	2	
		2
	2	
		2
1	1	
	5	

Abb. 16. Muster aus beispieldaten5.txt (in rot und blau), mit umgebenden Batterien in schwarz.

10. Zwischen einer menschlichen Spielerin/einem menschlichen Spieler und dem oben vorgestellten Verfahren gibt es viele Ähnlichkeiten. Beide benötigen tendenziell mehr Zeit, wenn von einer Batterie aus viele andere Batterien erreicht werden können (und trotzdem nur eine/wenige Batteriefeldabfolgen existieren). Wenn eine menschliche Spielerin/ein menschlicher Spieler jedoch genau verstanden hat, wie Batterien entladen werden können und welche Sonderfälle möglich sind, ist zu erwarten, dass die Spielerin/der Spieler die Spielsituation besser löst als der Algorithmus. Mit „besser“ ist nicht unbedingt „schneller“ gemeint, sondern weniger „Herumprobieren“ und mehr gezieltes Suchen der Lösung. Ein Mensch kann Kausalitäten besser erkennen als der Algorithmus und auf diese Weise eher Wege ausschließen. Für besonders schwere Spielsituationen ist daher zu erwarten, dass, relativiert zur „Rechenleistung“, ein Mensch schneller eine Lösung findet als der Algorithmus.
    Die Ähnlichkeit bedeutet jedoch nicht, dass die Schwierigkeit einer (für menschliche Spieler*innen) generierten Spielsituation daran beurteilt werden sollte, wie schnell sie vom Algorithmus gelöst wird. Für den Algorithmus spielt im Gegensatz zum menschlichen Spieler beispielsweise eine Rolle, wie gut die Heuristiken funktionieren. Für zwei Spielsituationen, die für Menschen ungefähr gleich schwer sind, kann der Algorithmus daher unterschiedlich viel Zeit benötigen.

Das Grundgerüst für die Spielfeldgenerierung bildet ein einfacher Algorithmus, der Batterien in der Reihenfolge auf ein Spielbrett legt, in der sie vom Roboter zur Lösung der Spielsituation besucht werden sollten. Die Anzahl der Felder, auf denen eine Batterie liegen soll, wird als Parameter `BATTERY_COUNT` vorgegeben. Das Spielbrett hat keine Größenbeschränkung (dennoch könnte eine Beschränkung einfach festgelegt werden), stattdessen ergibt sich die Größe am Ende, wenn alle Batterien platziert wurden. Alle Batteriefelder, die mit der aktuellen Batterie erreichbar sind (d.h. die aktuelle Batterie wird beim Betreten leer) und auf die als nächstes eine Batterie gelegt werden könnte, werden wieder mit dem Lee-Algorithmus ermittelt, indem vom aktuellen Batteriefeld aus Feldern die noch verbleibende Ladung zugeordnet wird. Ist die verbleibende Ladung auf einem Feld mod 2 = 0, dann kann auf dieses Feld als nächstes eine Batterie gelegt werden (außer in einem Sonderfall, s.u.):

0	1	2	3	2	1	0	
1	2	3	4_2	3	2	1	0
2	3	4_2	5_3	**0**	3	2	1
3	4_2	5_3	**6**	5_3	4_2	3	2
2	3	4_2	5_3	**0**	3	2	1
1	2	3	4_2	3	2	1	0
0	1	2	3	2	1	0	
	0	1	2	1	0		

Abb. 17. Anwendung des Lee-Algorithmus, um Felder ausgehend von der Batterie mit Ladung 6 zu ermitteln, auf die als nächstes eine Batterie gelegt werden kann. Die beiden Batterien mit Ladung 0 stellen Felder dar, auf denen bereits eine Batterie liegt. Die Bedeutung der tiefgestellten Zahlen wird im Text erklärt.

Die Felder in Abb. 17, die mit 0, 2 oder 4 markiert sind, kommen als Felder für die nächste Batterie infrage. Die Wege zu diesen Feldern sind entweder 2, 4 oder 6 Felder lang. Auf den Wegen der Länge

6 wird keine Ladung verfallen gelassen, auf Wegen der Länge 4 ist es auf jeden Fall möglich, ausreichend Ladung verfallen zu lassen, um die Batterie zu entladen. Auf den Wegen zu den mit 4 markierten Feldern mit Länge 2 kann jedoch nicht ausreichend Ladung verfallen gelassen werden. Zu allen grün eingefärbten Vieren existiert zusätzlich ein Weg der Länge 4, und daher kommen diese Felder trotzdem infrage. Diese Felder werden gefunden, indem der Algorithmus zu Beginn auch bereits besuchte Felder mit einer zweiten restlichen Ladung nochmals besucht und markiert (in Abb. 17 tiefgestellt). Die anfängliche Ladung (hier 6) sei a. Bis zur Ladung

$$a - \begin{cases} 4, \text{falls } a \bmod 2 = 0 \\ 3, \text{falls } a \bmod 2 = 1 \end{cases}$$

werden bereits besuchte Felder nochmals besucht, hier also bis zur Ladung 2. Dieser Wert ist genau um zwei kleiner als die restliche Ladung, bei der das Feld nicht sicher als nächstes besucht werden kann (ein Feld kann nicht sicher als nächstes besucht werden, wenn die restliche Ladung $a - 2$ bei geradem a bzw. $a - 1$ bei ungeradem beträgt). In Abb. 17 ist zwar auch die rot eingefärbte Vier so erreichbar, dass Ladung 2 übrigbleibt. Dabei müsste aber das Feld betreten werden, auf das eine Batterie gelegt werden soll, (nämlich das mit der 4) – und damit ist der Weg nicht mehr möglich. Zu allen anderen Batterien existiert ein Weg, mit dem das Feld mit restlicher Ladung 4 nicht betreten wird (beispielsweise `ulur` für das Feld unter der Batterie mit Ladung 6). Eines der möglichen nächsten Felder wird nach bestimmten Kriterien und „mit einer Prise Zufall“ ausgewählt, siehe Abschnitt 1.5.1. Zuerst im Folgenden das Grundgerüst für den Algorithmus:

1. Wähle eine zufällige Batterieladung[16] für die Roboterbatterie und eine beliebige Position für das Roboterstartfeld (die Position des Roboterstartfelds ist nicht relevant, weil das Spielfeld ohnehin keine Grenzen hat).
2. Suche per Maze-Routing alle Felder, die vom aktuellen Batteriefeld aus (zu Beginn das Roboterstartfeld) erreicht werden können (auf dem Weg zum Feld und auf dem Feld selbst darf noch keine Batterie liegen[17] und mögliche neue Batteriefelder dürfen außerdem nicht bereits als Weg genutzt werden).
3. Wähle aus allen gefundenen Feldern eines aus.
4. Markiere den Weg, der zum neuen Feld führt. Diese Felder dürfen nur für andere Wege, nicht aber für neue Batterien genutzt werden.
5. Wähle für das neue Feld eine zufällige Ladung.
6. Lege auf dieses Feld eine Ersatzbatterie mit der Ladung aus Schritt 5.
7. Wenn die Anzahl aller abgelegten Batterien noch nicht der gewünschten entspricht: Gehe zurück zu Schritt 2.

Dieses Grundgerüst berücksichtigt nicht, dass ein Batteriefeld doppelt besucht werden kann. Ein weiterer Parameter `PROBABILITY_MARK_AS_MULTI_VISIT` gibt an, mit welcher Wahrscheinlichkeit ein Batteriefeld mehrfach besucht werden soll. Für jede neu hinzugefügte Ersatzbatterie wird mit dieser Wahrscheinlichkeit per Zufall bestimmt, ob das nächste Batteriefeld doppelt besucht werden soll. Ist das der Fall, werden die möglichen Felder für das nächste Batteriefeld nicht mit der Ladung aus Schritt 5 berechnet, sondern die Ladung aus Schritt 5 reduziert um eine Zufallszahl wird genutzt, um mögliche nächste Felder in Schritt 2 zu bestimmen. Eine Batterie mit der Ladung aus Schritt 5 wird trotzdem unverändert auf das Feld gelegt. Dadurch bleibt beim Betreten des Feldes für die nächste Batterie

[16] Die Batterieladungen können beispielsweise Zufallszahlen von 1 bis 10 sein, wobei jede Ladung mit gleicher Wahrscheinlichkeit auftritt. Es ist aber auch denkbar, Ladungen mit einer Dreiecksverteilung auszuwählen: Die Batterieladung 1 tritt nur selten auf, die Ladung 3 am häufigsten und 11 ist die maximale Ladung, die ebenfalls selten auftritt. Diese Dreiecksverteilung wurde in der Implementierung gewählt. Übrigens ist es nur begrenzt sinnvoll, die Batterieladung basierend auf dem aktuellen Feld zu verändern, weil dadurch Hinweise auf den richtigen Weg entstehen.

[17] Die anfängliche Bordbatterie liegt nicht auf einem Feld.

etwas Ladung übrig. Diese nächste Batterie sollte nochmals besucht werden und wird als eine solche markiert.

Die bisherigen Schritte werden an einem Beispiel erläutert:

	8$_3$	
	5$_2$	

Abb. 18. Die Batterie mit Ladung 5 wurde auf einem Feld platziert. Die nächste Batterie soll doppelt besucht werden. Eine zufällige Zahl, hier 3, wird daher von der Ladung abgezogen und in Schritt 2 für die nächste Batterie werden neue Felder gesucht, wobei „so getan" wird, als ob die alte Batterie die Ladung 5-3=2 hätte (tiefgestellt). Als neues Batteriefeld wird das mit „8_3" beschriftete Feld ausgewählt. Auf diesem Feld liegt eine Batterie mit Ladung 8, gleichzeitig ist gespeichert, dass dieses Feld später nochmals besucht werden muss und dann eine Batterie mit Ladung 3 auf diesem Feld liegt.

Dadurch, dass sich durch dieses Verfahren an der Ladung der Batterien auf dem Feld nichts ändert, gibt die Ladung auch keinen Hinweis darauf, welche Batteriefelder in der Lösung mehrfach besucht werden müssen. Ein Hinweis könnte sein, dass wenn die reduzierte Ladung nicht dieselbe Parität besitzt wie die ursprüngliche Ladung, wie das in Abb. 18 der Fall ist, sich dann eine Batterie in der Nähe einer anderen Batterie befindet, die eigentlich nicht erreichbar wäre. In Abb. 18 liegt die Batterie mit Ladung 8 in der Nähe der Batterie mit Ladung 5, obwohl die Batterie mit Ladung 5 die Batterie mit Ladung 8 nicht direkt erreichen kann, sondern Ladung übrig bleiben würde. Sobald mehr Batterien generiert werden, fällt dies jedoch nicht mehr auf, da es dann wohl Batterien gibt, mit denen sich die Batterie mit Ladung 8 direkt erreichen lässt. Zudem wird dies durch die Auswahl des nächsten Feldes verhindert, weil als nächstes Feld mit größerer Wahrscheinlichkeit eines ausgewählt wird, das von vielen anderen Batterien erreicht werden kann (siehe Abschnitt 1.5.1).

Um ein Feld, das doppelt besucht werden soll, erneut zu besuchen, wird es vom Maze-Routing-Verfahren in Schritt 2 des obigen Algorithmus auch als mögliches nächstes Feld gespeichert, obwohl sich bereits eine Batterie auf dem Feld befindet. Bei der Auswahl des nächsten Feldes (siehe Abschnitt 1.5.1) werden Felder, die nochmals besucht werden müssen, mit einer größeren Wahrscheinlichkeit ausgewählt.

Es kann vorkommen, dass die vorgegebene Anzahl an Batterien bereits auf das Feld gelegt wurde, es aber noch Batterien gibt, die der Algorithmus nochmals besuchen müsste. Für diesen Fall gibt es mehrere Lösungsmöglichkeiten:

- Es werden mehr Batterien als vorgegeben generiert, solange, bis es keine Batterien mehr gibt, die mehrfach besucht werden müssen. Diese Möglichkeit ist am besten dazu geeignet, keine Hinweise über mehrfach besuchte Batterien zu geben.
- Dem ersten Punkt ähnlich, ist es ebenfalls möglich, solange bereits platzierte Batterien vom Feld in der umgekehrten Reihenfolge, in der sie generiert wurden, wegzunehmen, bis es keine Batterien mehr gibt, die nochmals besucht werden müssen.
- Soll die vorgegebene Batterieanzahl genau eingehalten werden, kann für jede noch nicht mehrfach besuchte Batterie die Ladung der Batterie reduziert werden, durch die Ladung beim Betreten der mehrfach zu besuchenden Batterie übrig blieb.[18] Die Ladung wird auf den Wert reduziert, der für die Suche nach dem nächsten Batteriefeld genutzt wurde. Diese Möglichkeit

[18] Meistens wird die Ladung derjenigen Batterie reduziert, die direkt vor der mehrfach zu besuchenden Batterie generiert wurde. Es ist jedoch auch möglich, dass die Ladung, die für die Suche nach dem nächsten Feld reduziert wurde, nicht von der zuvor generierten Batterie stammt, sondern von einer anderen Batterie. Dies ist der Fall, wenn die zuvor generierte Batterie selbst erneut besucht wurde und hierbei entschieden wurde, dass die folgende Batterie mehrfach besucht werden soll.

wurde in der Implementierung umgesetzt. Die vorgegebene Batterieanzahl wird eingehalten, dafür gibt es aber auch weniger „Mehrfachbesuche“ als durch die Wahrscheinlichkeit `PROBABILITY_MARK_AS_MULTI_VISIT` vorgegeben. Die Batterien, mit denen eine Batterie mehrfach besucht wird, haben dadurch eher eine höhere Ladung als andere. Dies betrifft jedoch nicht viele Batterien und es ist davon auszugehen, dass die Spielsituation dadurch nicht leichter wird.

1.5.1 Nächstes Feld auswählen

Das nächste Feld wird nach verschiedenen Kriterien ausgewählt. Wie in Punkt 8 beschrieben, wird ein Feld schwieriger, wenn Batterien gut verbunden sind. Das nächste Feld wird daher so gewählt, dass viele bereits generierte Batterien dieses Feld erreichen können. Dazu existiert ein zweites Spielfeld, dessen Felder für jedes Feld auf dem echten Spielfeld einen lokalen Schwierigkeitsgrad in Form einer Ganzzahl speichern. Der Schwierigkeitsgrad eines Feldes gibt an, mit wie vielen bereits platzierten Batterien eine Batterie mit Ladung 0 auf dieses Feld gelegt werden kann. Zu Beginn hat jedes Feld den Schwierigkeitsgrad 1. Sobald eine neue Batterie auf ein Feld platziert wird, wird der Schwierigkeitsgrad aller in Schritt 2 ermittelten Felder, die von der neuen Batterie aus erreicht werden können, um 1 erhöht.[19,20] Werden nächste Felder gesucht, wird für jedes mögliche nächste Feld der Schwierigkeitsgrad ermittelt. Aus diesen Feldern könnte nun eines ausgewählt werden, das den höchsten Schwierigkeitsgrad hat. Um jedoch auch Felder auszuwählen, die vielleicht eine niedrigere Schwierigkeit haben, erhält jedes Feld eine dem Schwierigkeitsgrad des Feldes entsprechende Gewichtung, mit der zufällig eines der Felder ausgewählt wird. Je nachdem, wie die Gewichtungen berechnet werden, sind die Batterien besser oder schlechter verbunden. Die Berechnung der Gewichtung aus der Schwierigkeit d (difficulty) sei definiert durch $s(d)$. Gilt $s(d) = 1$, sind die Batterien am schlechtesten verbunden und eine Lösung kann schnell gefunden werden. Mit beispielsweise $s(d) = d!$ erhalten höhere Schwierigkeiten deutlich mehr Gewichtung. Sind die Batterien allerdings zu gut verbunden, kann die Spielsituation wieder leichter zu lösen sein, weil damit auch die Zahl der möglichen Batteriefeldabfolgen steigt. Welche Berechnung geeignet ist, wird in Abschnitt 1.5.3 besprochen.

Bei der Gewichtungsberechnung wird auch berücksichtigt, ob zum Erreichen des Feldes Ladung verfallen gelassen werden muss oder nicht. Felder, die mit einem Weg erreicht werden können, bei dem keine Ladung verfallen gelassen wird, heißen schnellste („fastest“) Felder. Alle anderen Felder sind normale („normal“) Felder; um sie zu erreichen, könnte die aktuelle Batterie auch weniger Ladung haben. Tendenziell werden mehr Lösungen möglich, wenn viel Ladung verfallen gelassen wird. Daher ist es sinnvoll, normalen Feldern eine geringere Gewichtung als schnellsten Feldern mit derselben Schwierigkeit zuzuweisen. Auch dazu siehe Abschnitt 1.5.3.

Befinden sich unter den möglichen nächsten Feldern auch Felder, die nochmals besucht werden müssen, wird zuerst entschieden, ob eines dieser Felder als nächstes besucht wird oder nicht. Eine Möglichkeit ist, ein nochmals zu besuchendes Feld „bei der ersten Gelegenheit“ zu besuchen. Besser ist, ein nochmals zu besuchendes Feld nur mit einer gewissen Wahrscheinlichkeit auszuwählen (Parameter `PROBABILITY_DO_MULTI_VISIT`; in Abgrenzung zu `PROBABILITY_MARK_AS_MULTI_VISIT` wird mit dieser Wahrscheinlichkeit nicht entschieden, *ob* ein Feld doppelt besucht wird, sondern *wann*). Je größer

[19] In der Implementierung wird für ein erreichbares Feld nicht 1 zum Schwierigkeitsgrad addiert, sondern 4, wenn auf dem Weg zum Feld keine Ladung verfallen gelassen wird, ansonsten wird eine Zufallszahl von 0 bis 2 addiert (inklusive 0 und 2). Dadurch erhalten Felder, die ohne Ladung verfallen zu lassen erreicht werden können, einen höheren Schwierigkeitsgrad. Dies ist sinnvoll, da bei der Auswahl des nächsten Feldes ebenfalls Felder bevorzugt werden, zu denen keine Ladung verfallen gelassen wird (z.B. mit Parameter `PROBABILITY_TAKE_FASTEST` aus Abschnitt 1.5.3 oder der unterschiedlichen Gewichtungsberechnung für schnellste und normale Felder, siehe nächster Absatz).

[20] Es kann natürlich vorkommen, dass die Schwierigkeit auf einem Feld mit neu generierten Batterien wieder abnimmt, weil bestimmte Wege dadurch nicht mehr möglich sind. Dies wird nicht berücksichtigt.

diese Wahrscheinlichkeit, desto weniger Batterien werden besucht, bevor ein nochmals zu besuchendes Feld erneut besucht wird.

Eine interessante Situation zeigt sich, wenn die Felder betrachtet werden, die von Batterien mit gerader und ungerader Ladung aus erreicht werden können:

0	1	0	1	0
1	0	1	0	1
0	1	**A**	1	0
1	0	1	0	1
0	1	0	1	0

Abb. 19. Gilt **A** mod 2 = 0, können alle mit 0 beschrifteten Felder erreicht werden. Gilt **A** mod 2 = 1, können alle mit 1 beschrifteten Felder erreicht werden (unter der Voraussetzung, dass **A** groß genug ist).

Wenn Batterien nur gerade Ladungen haben, sind die Batterien offenbar besonders gut verbunden, weil sich dadurch Batterien nur auf den grau ausgefüllten Feldern befinden können.[21] Diese Erkenntnis kann für die Generierung besonders ausgefallener Spielsituationen genutzt werden.

Es kann vorkommen, dass kein nächstes Feld erreicht werden kann, weil alle erreichbaren Felder durch bereits gelegte Batterien oder Wege besetzt sind. Das Auftreten dieses Falles kann eingeschränkt werden, indem vorzugsweise bereits als Weg markierte Felder erneut als Weg genutzt werden und so weniger freie Felder als Weg markiert werden (siehe auch Abschnitt 1.5.2). Außerdem ist es möglich, beispielsweise per Backtracking ein anderes Feld zu suchen oder die Batterieladung so zu verändern, dass ein nächstes Feld erreicht werden kann. Da dieser Fall nur selten und mit hoher Batterieanzahl auftritt, wurde dies nicht implementiert.

1.5.2 Maze-Routing-Optimierung

Das Maze-Routing-Verfahren sollte zu einem Feld verschiedene Wege finden. Wurde ein Feld ausgewählt, kann danach einer der Wege ausgewählt werden, die zu diesem Feld führen. Als am besten geeignet wird der Weg angesehen, der möglichst bereits auf dem Spielfeld als Wege markierte Felder nutzt; die Anzahl der Felder, die neu als Weg markiert werden müssen, sollte also möglichst gering sein. Eine Möglichkeit wäre, bei der Rückverfolgung des Wegs nicht nur einen möglichen Weg, sondern alle zu suchen.[22] Für höhere Ladungen steigt die Anzahl dieser möglichen Wege jedoch stark an, sodass für mehr Effizienz nur ein Weg berechnet wird. Stattdessen wird die Reihenfolge, in der neue Felder mit der nächstniedrigeren verbleibenden Ladung besucht werden, verändert. Die neuen Felder werden gemischt und dann mit einem stabilen Sortierverfahren so sortiert,[23] dass Felder, die bereits als Weg markiert wurden, zuerst besucht werden, und erst danach Felder, für die ein neues Feld als Weg markiert werden müsste. Auf diese Weise wird jedem Feld nur genau ein Weg zugeordnet. Es ist nicht sichergestellt, dass dadurch vom Maze-Routing-Verfahren der Weg gefunden wird, bei dem am wenigsten Felder neu als Weg markiert werden müssen, aber diese Methode ist deutlich effizienter.

1.5.3 Auswahl geeigneter Parameter und Schwierigkeitsmaß

Im Folgenden eine Liste aller Parameter und Funktionen, die verändert werden können:

- `BATTERY_COUNT`: Anzahl der Batterien, die generiert werden sollen.

[21] Es sind Ausnahmen möglich, wenn Felder mehrfach besucht werden.
[22] In der Implementierung wird der Einfachheit halber der Weg zu einem Feld direkt gespeichert, sodass keine Rückverfolgung nötig ist. Das Problem, dass unter Umständen sehr viele Wege möglich sind, existiert in gleicher Weise bei dieser Methode.
[23] Der Lee-Algorithmus ist so umgesetzt, dass beim Markieren der Felder mit Ladung c die neuen Felder mit verbleibender Ladung $c - 1$ in einer Liste gespeichert werden. Diese Liste wird gemischt und sortiert.

- `NEW_CHARGE`: Funktion, die eine neue, zufällige Ladung zufällig zurückgibt. Diese nutzt die in Fußnote 16 beschriebene Dreiecksverteilung.
- `PROBABILITY_MARK_AS_MULTI_VISIT`: Die Wahrscheinlichkeit, mit der für die nächste Batterie „so getan“ wird, als wäre die Ladung bei der Suche nach möglichen nächsten Feldern geringer, sodass das ausgewählte Feld doppelt besucht werden kann.
- `PROBABILITY_DO_MULTI_VISIT`: Wenn nochmals zu besuchende Felder als nächste Felder erreicht werden können, wird mit dieser Wahrscheinlichkeit eines dieser Felder als nächstes Feld ausgewählt. Je kleiner die Wahrscheinlichkeit, desto mehr Batterien werden besucht, bevor ein Feld erneut besucht wird.
- `CALC_FASTEST_WEIGHT`: Funktion, die die Gewichtung für ein mögliches nächstes, schnellstes Feld berechnet.
- `CALC_NORMAL_WEIGHT`: Funktion, die die Gewichtung für ein mögliches nächstes, normales Feld berechnet.
- `PROBABILITY_TAKE_FASTEST`: Nachdem eine der möglichen Schwierigkeiten für das nächste Feld ausgewählt wurde, wird eines der Felder mit dieser Schwierigkeit ausgewählt. Schnellste Felder werden dabei bevorzugt. Ist eines der möglichen Felder ein schnellstes, wird es mit dieser Wahrscheinlichkeit ausgewählt, ansonsten wird ein normales Feld ausgewählt (wenn es keine normalen Felder gibt, wird immer ein schnellstes ausgewählt).

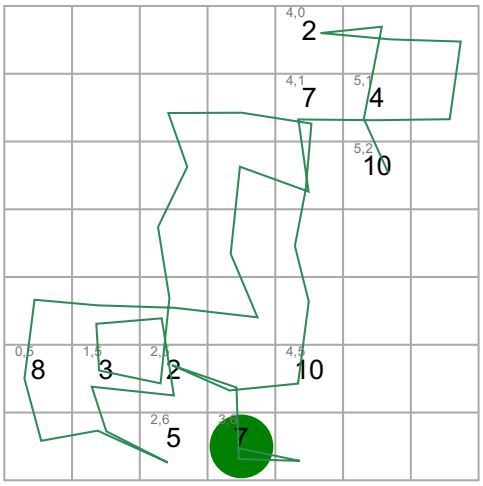
Von diesen Parametern beeinflussen einige mehr und einige weniger die Schwierigkeit des generierten Feldes. In diesem Abschnitt werden die Auswirkungen der Parameter auf die generierten Spielfelder untersucht. Mit den Ergebnissen wird ein Schwierigkeitsmaß definiert, das zwischen „Anfänger*innen“, „Fortgeschrittenen“ und „Expert*innen“ unterscheidet.

Für `BATTERY_COUNT` gilt grundsätzlich: Je größer der Wert, desto aufwendiger ist die Spielsituation zu lösen.

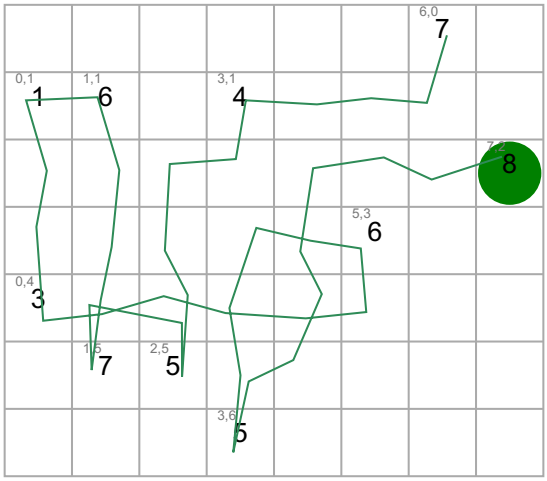
Der Einfluss von `NEW_CHARGE` auf die Schwierigkeit ist gering. Eine Spielsituation wird mit höheren Batterieladungen nicht zwingend schwieriger: Werden Batterien mit höheren Ladungen generiert, steigt auch die Wahrscheinlichkeit, dass es mehr als eine Lösung (verschiedene Batteriefeldabfolgen) gibt. Für alle Schwierigkeiten werden Batterieladungen daher auf dieselbe Weise – mit der Dreiecksverteilung aus Fußnote 16 – generiert.

Die Werte für `PROBABILITY_MARK_AS_MULTI_VISIT` und `PROBABILITY_DO_MULTI_VISIT` sind größtenteils frei wählbar. Für besonders leichte Spielfelder kann `PROBABILITY_MARK_AS_MULTI_VISIT` auf 0 gesetzt werden. Grundsätzlich macht ein höherer Wert für `PROBABILITY_MARK_AS_MULTI_VISIT` die Spielsituation schwieriger, auch wenn es Lösungen gibt, die nicht die vorgesehenen Batteriefelder

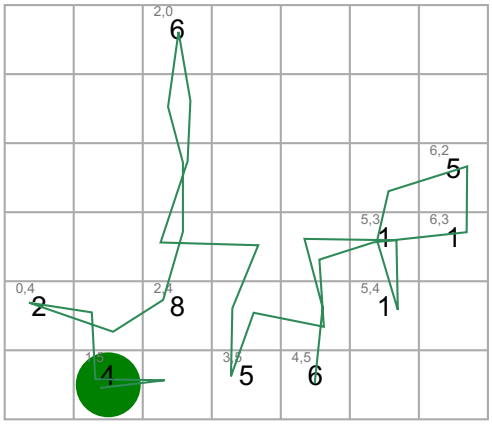
mehrfach besuchen. Folgendes recht anspruchsvoll zu lösendes Spielfeld wurde mit `PROBABILITY_MARK_AS_MULTI_VISIT = 90%` generiert.[24]



Anders sieht das Spielfeld aus, wenn eine Wahrscheinlichkeit von 50% gewählt wird:



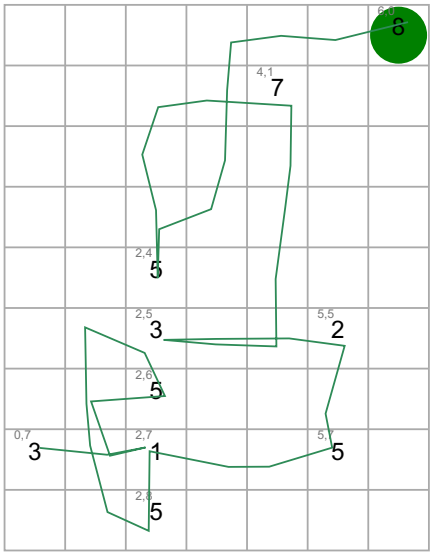
Für beide Spielfelder war `PROBABILITY_DO_MULTI_VISIT` auf 60% gesetzt. Im Folgenden nochmals das Spielfeld mit `PROBABILITY_MARK_AS_MULTI_VISIT = 90%`, nur für `PROBABILITY_DO_MULTI_VISIT` wurde stattdessen ein Wert von 40% gewählt:



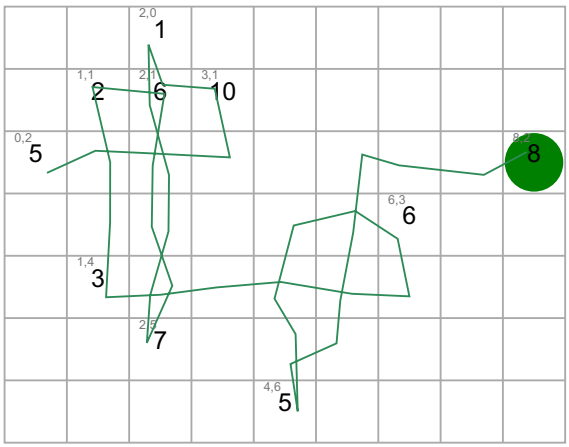
[24] Die gezeigten Abbildungen wurden mit den Parametern für die Schwierigkeit „Fortgeschrittene*r“ generiert, nur die jeweils betrachteten Parameter wurden verändert. Der Startwert betrug immer 0. Auf den hier gezeigten Abbildungen ist die von der Generierung vorgesehene Lösung eingezeichnet. Der Weg zeigt nicht, wenn der Roboter Ladung verfallen lässt. Der Linienverlauf ist randomisiert, damit der Weg auch bei Überschneidungen erkennbar ist (teilweise wurde der Verlauf manuell optimiert).

Vom Generierungsalgorithmus sind für die Lösung dieser Spielsituation deutlich weniger mehrfach besuchte Batterien vorgesehen.

Wie bereits angesprochen, wird das Feld leichter, wenn jedes Feld die gleiche Gewichtung enthält, wenn also `CALC_FASTEST_WEIGHT` und `CALC_NORMAL_WEIGHT` immer denselben Wert zurückgeben ($s(d) = 1$). Die folgende Spielsituation wurde mit dieser Gewichtung generiert.

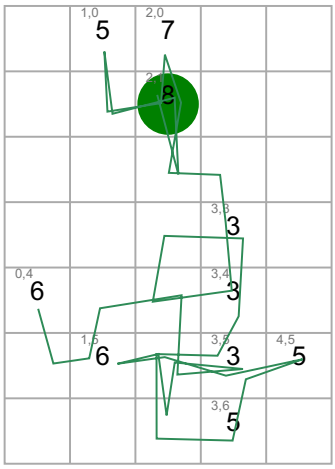


Von einer Batterie sind in dieser Spielsituation durchschnittlich 2,3 andere Batterien erreichbar. Ein Durchschnitt von 3,1 wird in folgender Spielsituation erzielt; zur Generierung wurde bevorzugt eines der Felder mit sehr hohem Schwierigkeitsgrad ausgewählt ($s(d) = d!$ für `CALC_FASTEST_WEIGHT` und `CALC_NORMAL_WEIGHT`):

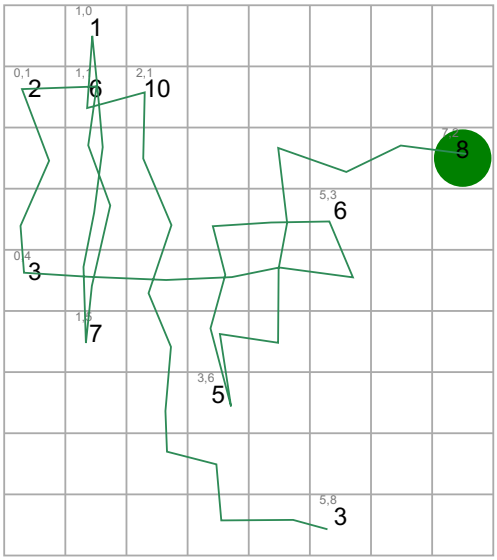


Mit solch einer hohen Gewichtung wird jedoch fast immer auf das Feld mit der höchsten Schwierigkeit als nächstes eine Batterie gelegt. Für etwas Abwechslung sollte die Gewichtung daher nicht allzu hoch sein, sodass auch gelegentlich Felder mit niedrigerem Schwierigkeitsgrad ausgewählt werden und die Batteriefelder sich besser auf dem Spielbrett verteilen. Außerdem sollten schnellste Felder eine größere Gewichtung erhalten als kleine. Für eine „normale“ Schwierigkeit wird daher `CALC_FASTEST_WEIGHT` durch $s(d) = d^2$ und `CALC_NORMAL_WEIGHT` durch $s(d) = d$ definiert.

Der Parameter `PROBABILITY_TAKE_FASTEST` sollte so gewählt werden, dass nicht zu oft Ladung verfallen gelassen wird. Folgende Spielsituation wurde mit einem Wert von 10% generiert:



Dieser Wert ist offensichtlich zu niedrig, um ständiges „Verfallenlassen“ von Ladung zu verhindern. Die folgende Spielsituationen wurde mit einem Wert von 80% generiert.



Schwierigkeitsmaß

Die Definition eines Schwierigkeitsmaßes als eine Funktionsgleichung ist nur bedingt sinnvoll und wird der vielfältigen Möglichkeiten für die Parameter nicht gerecht. Stattdessen wird im Folgenden eine Unterteilung in drei verschiedene Schwierigkeitslevel vorgenommen. Generierte Spielfelder zu jedem Schwierigkeitsgrad finden sich im Abschnitt 3 „Beispiele“.

Für **ANFÄNGER*INNEN** ist es sinnvoll, dass

- wenige oder keine Batterien mehrfach besucht werden müssen. Werte zwischen 0 und 50% für `PROBABILITY_MARK_AS_MULTI_VISIT` und ähnliche Werte für `PROBABILITY_DO_MULTI_VISIT` sind sinnvoll.
- von einer Batterie aus nicht zu viele andere Batterien erreicht werden können. `CALC_FASTEST_WEIGHT` kann beispielsweise durch $s(d) = d$ definiert sein und `CALC_NORMAL_WEIGHT` durch $s(d) = 1$.
- nur wenige Batterien generiert werden.

Bei Spielsituationen für **FORTGESCHRITTENE** (diese Parameter wurden für die obigen Beispiele genutzt) sollten

- Felder häufig mehrfach besucht werden müssen. Werte um 70% für `PROBABILITY_MARK_AS_MULTI_VISIT` sind sinnvoll. Der Wert für `PROBABILITY_DO_MULTI_VISIT` ist nicht so entscheidend für die Schwierigkeit wie der für `PROBABILITY_MARK_AS_MULTI_VISIT`. Für die obigen Beispiele wurde ein Wert von 60% genutzt.
- Wie oben beschrieben, wird `CALC_FASTEST_WEIGHT` durch $s(d) = d^2$ und `CALC_NORMAL_WEIGHT` durch $s(d) = d$ definiert.

Spielsituationen für **EXPERTINNEN UND EXPERTEN** zeichnen sich durch folgende Parameterwerte aus:

- Sehr hoher Wert für `PROBABILITY_MARK_AS_MULTI_VISIT`, beispielsweise 90%.
- Funktionen für die `CALC_*_WEIGHT`-Parameter, die auch Felder mit niedrigerem Schwierigkeitsgrad gelegentlich zulassen (also z.B. keine Fakultäten). Geeignete Funktionen sind $s(d) = 2^d$ oder $s(d) = d^4$ (für `CALC_NORMAL_WEIGHT`). Für `CALC_NORMAL_WEIGHT` kann wie für Fortgeschrittene $s(d) = d$ gewählt werden, gerade bei exponentieller Gewichtung für schnellste Felder kann auch eine andere Gewichtung sinnvoll sein.

Diese Unterteilung soll Orientierung geben, welche Werte sich eher für welche Schwierigkeit eignen. Es ist nicht ausgeschlossen, dass einzelne Vorschläge für Parameter bei einer anderen Schwierigkeit ebenfalls sinnvolle Ergebnisse liefern.

2 Umsetzung

Die Algorithmen wurden in Python 3.8.1 implementiert. Aufgrund der Verwendung neuer Sprachelemente sind die Skripte nicht mit älteren Versionen kompatibel. Die Algorithmen im `random`-Modul können sich zwischen Python-Versionen ändern, weshalb mit den angegebenen Startwerten unter Umständen andere Spielsituationen generiert werden.[25] Die Drittanbietermodule `numpy`, `graphviz` und `svgwrite` werden benötigt (`pip3.8 install -r requirements.txt`).

Der Quellcode ist in drei Skripte aufgeteilt:

```
$ python3.8 aufgabe1a.py -h

usage: aufgabe1a.py [-h] [--print] [filename]

positional arguments:
  filename    Filename in directory 'beispieldaten' to process. If not specified,
processes stromrallye0.txt ... stromrallye5.txt

optional arguments:
  -h, --help  show this help message and exit
  --print     show output while doing backtracking

Saves graphviz-graphs to 'beispieldaten_svg' and solutions to 'solutions'
```

aufgabe1a.py kann als Kommandozeilenparameter ein Dateiname im `beispieldaten`-Ordner übergeben werden. Die Datei wird dann eingelesen, die Lösung berechnet und ausgegeben und eine die Lösung darstellende Vektorgrafik im Ordner `solutions` gespeichert. Wird keine Datei angegeben, werden die Beispieldaten berechnet. Die Option `--print` spezifiziert, dass Ausgaben zum Backtracking gemacht werden sollen. Die beiden erstellten Graphen werden in vier Dateien (zwei Graphviz-Dateien, zwei SVGs) im `beispieldaten_svg`-Ordner gespeichert.

`aufgabe1b.py`

Generiert zehn Spielbretter mit den Startwerten 0, …, 9. Die Spielfelder als SVGs werden im `generated_examples`-Verzeichnis gespeichert. Für jedes Spielfeld wird mit aufgabe1a.py eine Lösung gesucht und ebenfalls in die Vektorgrafik eingefügt. Die beiden erstellten Graphen werden im selben

[25] docs.python.org/3/library/random.html#notes-on-reproducibility

Verzeichnis gespeichert. Die Parameter aus Abschnitt 1.5.3 sind als gleichnamige Konstanten[26] implementiert und werden an die Hauptfunktion, `generate_stomrallye`, übergeben.

`util.py`

Diese Datei enthält keinen für die Lösung der Aufgabenstellung relevanten Code. In der Datei befinden sich:

- Die Klasse `Vector2`, die dazu geeignet ist, Punkte mit ganzzahligen Koordinaten im zweidimensionalen Raum darzustellen,
- die Funktion `save_svg`, die ein Spielfeld als SVG speichert und
- die Klasse `StromrallyeParser`, die eine Spielfelddatei einliest.

Das Dateiformat wurde erweitert, damit noch mehr Informationen in der Datei gespeichert werden können und beispielsweise der Ursprung der Koordinaten auf 0,0 statt 1,1 geändert werden kann. Von aufgabe1b.py werden standardmäßig Dateien im erweiterten Format gespeichert; wird die Konstante `EXTENDED_FILE_FORMAT` auf `False` geändert, werden die Dateien im vorgegebenen Format abgespeichert. Das erweiterte Format ist in `beispieldaten/README_EXTENDED_FORMAT.txt` erläutert.

2.1 Maze-Routing: Erstellung des ersten Graphen mit Distanzen

Die statische Methode `from_file` der Klasse `Robot` aus aufgabe1a.py liest eine Spielsituation ein und erstellt den ersten Graphen mit einem Aufruf der statischen Methode `from_battery_fields`, die ein `Robot`-Objekt zurückgibt. Mit der Methode `create_fake_batteries` wird der zweite Graph erstellt, in dem `find_way` nach einem Weg sucht. `from_battery_fields` erhält von `from_file` eine Liste aller Batteriefelder. Ein Batteriefeld wird durch die Klasse `BatteryField` dargestellt; die Klasse `Battery` repräsentiert eine Batterie, also einen Knoten im zweiten Graphen. Zu Anfang sind nur folgende Attribute eines `BatteryField`s gesetzt:

- `id`: Jedem Feld wird eine eindeutige Nummer zugeordnet, beginnend bei 0.
- `pos` speichert die Position auf dem Spielbrett (mit Ursprung 0,0; die Koordinaten der Beispieldaten werden entsprechend korrigiert).
- `initial_battery` speichert eine Instanz von `Battery`, die die Ladung darstellt, die sich zu Beginn auf diesem Feld befindet (gelb eingefärbte Knoten im zweiten Graphen).

Der Aufruf von `from_battery_fields` setzt folgende Attribute:

- `connections`: Eine Kante zwischen zwei `BatteryField`s wird durch ein Objekt der Klasse `BatteryFieldConnection` dargestellt. Dieses Attribut speichert eine Liste von `BatteryFieldConnection`s, die alle Kanten darstellen, die von diesem `BatteryField` wegführen. Ein Objekt von `BatteryFieldConnection` speichert das andere `BatteryField` (Attribut `other_field`), die Distanz zwischen den Batteriefeldern (`distance`), die Distanz eines Weges länger als 2 (`distance_greater_than_two`; ist -1, wenn es keinen längeren Weg gibt) und die zu `distance` und `distance_greater_than_two` gehörenden Wege als eine Liste von `Vector2`-Objekten: `way` und `longer_way`. Die letztgenannten zwei Attribute sind nur relevant, um am Ende einen Weg bestehend aus allen Feldern, die der Roboter betritt, mit der Methode `Robot.generate_full_way` zu generieren.
- `free_fields_around`: Eine Liste, die die freien zusammenhängenden Felder speichert. Erstes Element enthält ein freies Nachbarfeld (wenn ein solches existiert), zweites Element enthält ein Nachbarfeld des Nachbarfelds (wenn ein solches existiert). Die Liste enthält also entweder 0, 1 oder 2 Elemente, je nachdem, wie viele freie zusammenhängende Felder es gibt.

[26] Konstanten sind in Python per Konvention Variablen, deren Name aus Großbuchstaben besteht. Sie sind keine echten Konstanten, sondern die Werte können (sollten aber nicht) geändert werden.

- `small_way_lengths_without_longer_ways`: Eine Menge (`set`). Siehe Lösungsidee, „swlolw“ im Graphen.

Das Spielbrett wird durch ein numpy-Array `field` dargestellt, in dem jedes Feld, auf dem sich ein Batteriefeld befindet, den Wert `True` hat; alle anderen sind `False`. Auch das Roboterstartfeld hat den Wert `False`, sodass dieses Feld korrekterweise als frei betrachtet wird. Alle Batteriefelder werden nach Position sortiert (dies ist optional, wie in der Lösungsidee beschrieben) und befinden sich in der Liste `battery_fields`. Diese Liste ist gleichbedeutend mit dem Array b aus Abschnitt 1.2.1. Für ein Batteriefeld `battery_field` aus `battery_fields[:-1]` (also ohne das letzte Element; zum letzten Batteriefeld sind ohnehin alle Distanzen bereits berechnet worden) werden die Entfernungen zu jedem Batteriefeld `other_battery_field` aus `battery_fields[x+1:]` bestimmt, wobei `battery_field` das `x`-te Element aus `battery_fields` ist. Ob sich zwischen den zwei Feldern `battery_field` und `other_battery_field` ein einfacher Weg ohne Maze-Routing finden lässt, kann mit dem `field`-Array überprüft werden: Alle Elemente im Array, die auf dem Weg liegen, müssen `False` sein. Lässt sich ein Weg nicht einfach bestimmen, wird `other_battery_field.id` in einem zweiten Array `batteries_for_maze_routing` gespeichert. Ist ein Element in diesem Array nicht `-1`, handelt es sich um die ID eines Batteriefelds, zu dem ein Weg per Maze-Routing gefunden werden soll. Auf dieselbe Art werden `BatteryFieldConnection`s gespeichert, bei denen zwar schon eine `distance` ermittelt wurde, aber noch keine `distance_greater_than_two`. Da es jeweils zwei `BatteryFieldConnection`s gibt (in jede Richtung eine)[27], werden die zwei Connections in einem dritten Array `connections_with_small_distances` gespeichert, dessen Elemente Listen sind, die zwei `BatteryFieldConnection`s enthalten. Wird per Maze-Routing eine Entfernung größer als zwei gefunden, werden die Attribute `distance_greater_than_two` und `longer_way` entsprechend gesetzt. Dabei muss beachtet werden, dass das Attribut `longer_way` (genauso wie `way`) bei der einen `BatteryFieldConnection` in umgedrehter Reihenfolge gespeichert werden muss.

Die Implementierung des Maze-Routings weist keinerlei Besonderheiten auf; es wird ein weiteres numpy-Array `visited` benötigt, dessen Elemente `set`s sind. In den Mengen werden alle Distanzen gespeichert, mit denen das Feld bereits besucht wurde. Beim Markieren von Feldern mit Distanz `distance` werden in zwei `dict`s Positionen gespeichert, die beim nächsten Durchlauf die Distanz `distance+1` erhalten. Die `dict`s heißen `one_visit_positions` und `double_visit_positions`. Ersteres speichert Positionen von Feldern, die zum ersten Mal besucht werden, letzteres speichert Positionen von Feldern, die zum zweiten Mal besucht werden. `double_visit_positons` ist nur relevant, solange `distance<5`, weil danach keine Felder mehr mit zwei Distanzen markiert werden müssen.

Am Ende der Funktion werden alle `BatteryField`s durchgegangen und für jedes werden der Menge `small_way_lengths_without_longer_ways` Elemente hinzugefügt, wenn die Distanz einer Connection aus `battery_field.connections` kleiner als 3 ist und `distance_greater_than_two` nicht 3 oder 4 ist, wenn es also nicht immer möglich ist, (gerade) Ladung verfallen zu lassen. Zudem werden freie (Nachbar-)Nachbarfelder ermittelt (die zusammenhängenden freien Felder) und in `battery_field.free_fields_around` gespeichert und es wird eine Connection zum Batteriefeld selbst hinzugefügt, sofern die Länge von `free_fields_around` 1 (die Connection hat keinen `longer_way`) oder 2 (die Connection hat einen `longer_way`) ist.

2.1.1 Teleportationsfelder

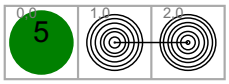
Die Methode `Robot.from_battery_fields` weicht leicht von obiger Beschreibung ab, um auch Teleportationsfelder zu berücksichtigen. Das numpy-Array `field` speichert nicht nur, ob sich ein Batterie- oder Teleportationsfeld auf einem Feld befindet, sondern auch die ID des Feldes. Dadurch kann beim

[27] Gilt nicht für das Roboterstartfeld. Das `other_field`-Attribut einer `BatteryFieldConnection` ist nie das Roboterstartfeld, weil vom Roboterstartfeld nur Kanten wegführen.

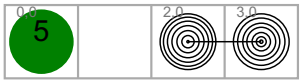
Befüllen von `free_fields_around` geprüft werden, ob sich auf einem Feld ein Teleportationsfeld befindet. Ein Teleportationsfeld wird durch ein Objekt der Klasse `TeleportationField` dargestellt. `BatteryField` und `TeleportationField` erben von `BaseField` und haben beide die Attribute `pos`, `id`, `connections` und `tp_connections`. Ein `TeleportationField` hat zusätzlich das Attribut `paired_field`, das auf das mit diesem Teleportationsfeld verbundenen Teleportationsfeld zeigt. Eine `BatteryFieldConnection` kann entweder auf ein `TeleportationField` oder ein `BatteryField` zeigen. Zeigt sie auf ein `BatteryField`, wird sie in `connections` gespeichert, ansonsten in `tp_connections`. Für `Robot.create_fake_batteries` ist nur das `connections`-Attribut interessant. Um eine `BatteryFieldConnection` zu einem `BaseField` hinzuzufügen, gibt es die Methode `add_connection`. Diese fügt die Connection der entsprechenden Liste an.

Die Wege, die für eine `BatteryFieldConnection` in den zwei Attributen gespeichert werden, erhalten ein weiteres Format: Bisher wurden die Wege immer als eine Liste von `Vector2`-Objekten gespeichert. Mit diesen Listen erstellt `Robot.generate_full_way` den vollständigen Weg. Sobald Ladung verfallen gelassen wird, kann `generate_full_way` normalerweise einfach die beiden letzten Positionen des Weges sooft wie nötig wiederholen, der Roboter bewegt sich also auf diesen beiden Feldern hin- und her. Mit Teleportationsfeldern wird, um Ladung verfallen zu lassen, aber nicht immer das letzte Stück des Weges wiederholt, sondern der Roboter sollte das erste Teleportieren sooft wiederholen (also sich zurückteleportieren lassen, dann die Teleportation wiederholen) wie nötig. Dafür wird die Abspeicherung des Weges in einem zweiten Format, einem Tupel, ermöglicht. Dieses Tupel besteht aus drei Elementen und jedes Element ist ein Teilweg. Der mittlere Teilweg, das zweite Element, besteht aus zwei Positionen und kann beliebig oft wiederholt werden.

Außerdem wird in einer Liste, die einen Weg dargestellt, `"tp"` zwischen zwei Positionen eingefügt, wann immer der Roboter teleportiert wird. Diese Art der Darstellung hat jedoch einen Nachteil: Es könnte nun sein, dass der Roboter für die zusammenhängenden freien Felder in `free_fields_around` teleportiert werden muss. Dadurch befinden sich aber unter Umständen mehr als zwei Elemente in der Liste (eigentlich sollten sich maximal zwei Elemente in der Liste befinden) und zusätzlich ist `generate_full_way` nicht bekannt, welche Elemente aus `free_fields_around` wiederholt werden sollen. Um dies zu lösen, werden gegebenenfalls `TeleportationField`-Objekte statt `Vector2`-Objekte in `free_fields_around` gespeichert. Für diesen Fall …



… wird in `free_fields_around` als erstes Element das `TeleportationField`-Objekte auf Feld 1,0 gespeichert. Auf das zweite Teleportationsfeld auf Feld 2,0 wird von `Robot.generate_full_way` durch das `paired_field`-Attribut zugegriffen. Damit `free_fields_around` trotzdem zwei Elemente enthält (um deutlich zu machen, dass eine beliebige Ladung „verfahren“ werden kann), ist das zweite Element `None`. Für den Fall, dass sich vor dem Teleportationsfeld ein freies Feld befindet …



… speichert das erste Element der Liste die Position des freien Feldes, hier also 1,0, und das zweite Element das Teleportationsfeld, hier auf Feld 2,0.

Die Liste aller Batteriefelder `battery_fields` wird um alle `TeleportationFields` erweitert. Die Teleportationsfelder erhalten genauso wie die `BatteryFields` IDs, wobei die IDs der Batteriefelder kleiner sind. Dadurch benötigt die Variable `active_battery_fields` (genutzt von `Robot.find_way`, siehe Abschnitt 2.3) möglichst wenig Bits.

Der Dijkstra-ähnliche Algorithmus ist in der lokalen Funktion[28] `get_teleportation_reachable_battery_fields` implementiert. Diesem wird ein Batteriefeld `start_field` übergeben, von dem aus Wege, die Teleportationsfelder nutzen, zu Batteriefeldern gefunden werden sollen. Wie beschrieben wird jeder Knoten durch zwei Knoten dargestellt, einem mit ungerader und einem mit gerader Distanz. Die Knoten mit gerader Distanz werden in dem `dict` `even_distances` gespeichert, die anderen in `odd_distances`. Schlüssel sind Batterie- oder Teleportationsfelder und die Werte sind ein Tupel bestehend aus der bisher zu diesem Feld gefundenen Distanz und dem zugehörigen Weg. Sobald ein Feld besucht wurde, wird es aus dem entsprechenden `dict` entfernt. Um herauszufinden, welches Feld als nächstes besucht wird, wird für beide `dicts` das Feld mit der kleinsten Distanz ermittelt (es wird keine Prioritätswarteschlange verwendet, für eine bessere Laufzeit wäre dies sinnvoll). Das Feld aus diesen beiden mit der kleineren Distanz wird als nächstes besucht. Bei der Relaxation wird das Feld in `even_distances` oder `odd_distances` aktualisiert, je nachdem, welche Parität die neue Distanz besitzt. Ansonsten weist die Implementierung keinerlei Besonderheiten auf.

2.2 Fakebatterien: Erstellung des zweiten Graphen

Der zweite Graph wird von der Methode `Robot.create_fake_batteries` erstellt. Jedes `BatteryField` speichert dazu im Attribut `batteries` Instanzen von `Battery`, die alle Batterien darstellen, die sich auf diesem Feld befinden können. `batteries` ist ein `dict`, das eine Ladung einem `Battery`-Objekt zuordnet. Auf diese Weise kann die Funktion `BatteryField.get_or_create_fake_battery`, die eine neue Batterie für das Batteriefeld erstellen soll und der die gewünschte Ladung übergeben wird, prüfen, ob ein neues `Battery`-Objekt instanziiert werden muss oder ob es die Batterie bereits gibt.[29] Das `Battery`-Objekt und ob es sich um eine neue Batterie handelt, wird zurückgegeben. Handelt es sich um eine neue Batterie, sucht `create_fake_batteries` für die neue Batterie rekursiv nach Batterien, zu denen eine Kante erstellt werden soll. Das rekursive Hinzufügen der Batterien und Kanten übernimmt die lokale Funktion `search_connections`. Dieser wird ein `Battery`-Objekt übergeben, zu dem wegführende Kanten erstellt werden sollen. Die wegführenden Kanten werden wie für `BatteryField` im Attribut `connections` gespeichert; diese sind Instanzen von `BatteryConnection`. Eine `BatteryConnection` speichert im Attribut `new_battery`, auf welche Batterie die Kante zeigt. Da jede `Battery` im Attribut `battery_field` das Batteriefeld speichert, zu dem die Batterie gehört, kann später das neue Batteriefeld per `BatteryConnection.new_battery.battery_field` ermittelt werden. Die aktive Batterie des Batteriefelds (gespeichert in `BatteryConnection.active_battery`) wird, wenn diese Kante beim Backtracking genutzt wird, auf `new_battery` gesetzt. Um später den vollständigen Weg generieren zu können, speichert eine `BatteryConnection` außerdem, aus welcher `BatteryFieldConnection` diese `BatteryConnection` „entstanden“ ist (Attribut `field_connection`) und ob dabei der normale Weg oder der länger als 2 genutzt wurde (Attribut `uses_long_field_connection` vom Typ `bool`). Auf diese Weise kann per `.field_connection.way` (wenn `not uses_long_field_connection`) bzw. `.field_connection.longer_way` (wenn `uses_long_field_connection`) von `Robot.generate_full_way` ermittelt werden, welchen Weg der Roboter nehmen sollte, um von einer Batterie zur anderen zu gelangen.

`search_connections` geht alle `BatteryFieldConnections` des Batteriefelds durch, zu dem die übergebene `Battery` `current_battery` gehört (Variable `field_connection`). Ist das `distance`-Attribut der Connection größer als `current_battery.charge`, kann diese Connection nicht genutzt werden. Ansonsten wird auf dem Batteriefeld `field_connection.other_field` eine `Battery` mit Ladung 0

[28] Lokale Funktionen sind in Python innerhalb einer anderen Funktion (hier `from_battery_fields`) definiert und können nur innerhalb dieser Funktion aufgerufen werden. Sie sind besonders gut zur Strukturierung des Quellcodes geeignet.

[29] Die anfänglich aktive Batterie ist zwar bereits im Attribut `initial_battery` des `BatteryFields` gespeichert, diese hat jedoch noch keine ID und wird daher mit der Information, dass die Batterie neu ist, zurückgegeben, sobald die Funktion mit der entsprechenden Ladung aufgerufen wird, und erst danach zu `batteries` hinzugefügt.

erstellt, wenn `field_connection.distance % 2 == current_battery.charge % 2`, wenn beide also dieselbe Parität haben. Zusätzlich muss entweder `current_battery.charge` kleiner als 3 sein, oder die `field_connection` muss einen Weg länger als 2 besitzen und es muss gelten `field_connection.distance_greater_than_two <= current_battery.charge`. Unabhängig von der Parität ist es immer möglich, eine Verbindung zu einer Batterie mit Ladung `current_battery.charge - field_connection.distance` auf dem Feld `field_connection.other_field` zu erstellen. Außerdem müssen Kanten entsprechend des `field_connection.other_field.small_lengths_without_longer_ways`-Attributs erstellt werden. Sollte zum Erreichen des anderen Batteriefelds mehr Ladung als 2 „verbraucht“ werden, muss wieder die `distance_greater_than_two` ausreichend klein sein.

Die `search_connections`-Funktion wird für jede anfängliche Batterie (=Ersatzbatterie) für jedes Batteriefeld aufgerufen. Es kann sein, dass eine Batterie gleichzeitig Fake- und Ersatzbatterie ist. Daher kann es vorkommen, dass die Connections für eine Ersatzbatterie bereits berechnet wurden und deshalb nicht nochmals berechnet werden müssen.

In der `create_fake_batteries`-Methode werden auch alle `BatteryConnection`s wie in der Lösungsidee beschrieben nach verschiedenen Kriterien sortiert. Außerdem erhält jedes Batteriefeld ein `neighbors`-Attribut. Dieses ist eine Menge, die alle Batteriefelder enthält, die mit Batterien auf diesem Batteriefeld erreicht werden können. Dies ist wichtig, um den Zusammenhang des Graphen zu überprüfen, siehe Abschnitt 1.3.2.

2.3 Backtracking

`Robot.find_way` implementiert das beschriebene Backtrackingverfahren. `Robot.find_way` hat zwei lokale Funktionen: `is_graph_connected`, um den Zusammenhang des Graphen zu überprüfen, und `find_way_backtracking`. Letztere ruft sich selbst rekursiv auf und erhält als Parameter `current_battery`, die Batterie, von der aus Wege gesucht werden sollen, und `new_battery`. Es gilt `current_battery.battery_field == new_battery.battery_field`. `new_battery` ist die Batterie, die der Roboter auf das Batteriefeld legt, während `current_battery` die Batterie ist, die er aufnimmt. `find_way` gibt das Ergebnis eines Aufrufs von `find_way_backtracking` mit den Argumenten `current_battery` = anfängliche Roboterbatterie und `new_battery` = Batterie mit Ladung 0 auf dem Roboterstartfeld zurück.

Im Scope von `find_way` sind zwei Variablen vom Typ `int` definiert, die die aktiven Batterien (`batteries`) und die aktiven Batteriefelder (`active_battery_fields`) speichern. In `batteries` ist das x. Bit gesetzt, wenn die Batterie mit ID x gerade die aktive Batterie im zu der Batterie gehörenden Batteriefeld ist. Batterien mit Ladung 0 benötigen keine ID, weil in dem Fall, dass die aktive Batterie eines Batteriefelds diejenige mit Ladung 0 ist, einfach keines der Bits gesetzt ist, die zu Batterien des Batteriefelds gehören. Bei einem Aufruf der `find_way_backtracking`-Funktion wird der Wert von `batteries` verändert. Das zu `current_battery` gehörende Bit wird auf 0 gesetzt und das zu `new_battery` – sofern `new_battery.charge != 0` – gehörende Bit auf 1. Entsprechend wird auch das Attribut `current_battery.battery_field` (oder `new_battery.battery_field`) auf `new_battery` gesetzt. Wird kein Weg gefunden, werden diese Änderungen rückgängig gemacht, und zwar bevor die Funktion `None` zurückgibt, um anzuzeigen, dass kein Weg gefunden wurde. `find_way_backtracking` kann mit der `batteries`-Variablen herausfinden, ob alle Batteriefelder besucht wurden (d.h. die aktive Batterie jedes Batteriefelds die Ladung 0 besitzt), nämlich dann, wenn `batteries == 0` gilt. In diesem Fall wurde ein Weg gefunden, sofern es das Batteriefeld von `current_battery` zulässt, die Ladung `current_battery.charge` zu „verfahren“. Dies wird mit einem Vergleich der Ladung mit der Länge von `current_battery.battery_field.free_fields_around` geprüft:

- Ist `current_battery.charge == 1`, ist die Länge von `free_fields_around` irrelevant: Der Roboter kann sich auf ein beliebiges Nachbarfeld bewegen, woraufhin alle Batterien leer sind.
- Ist `current_battery.charge == 2`, muss `free_fields_around` mindestens ein Element enthalten.
- Ist `current_battery.charge > 2`, muss `free_fields_around` zwei Elemente (=Maximum) enthalten.

`batteries` wird außerdem für den Cache benötigt. Der Cache ist ein `set`, das wiederum im Scope von `find_way` definiert ist. Bei einem Aufruf von `find_way_backtracking` wird geprüft, ob sich das Tupel `(batteries, current_battery)`[30] bereits in der Menge befindet. Ist dies der Fall, wurde `find_way_backtracking` bereits mit diesen zwei Parametern aufgerufen und es wurde kein Weg gefunden, weshalb direkt `None` zurückgegeben wird. Ansonsten wird das Tupel zur Menge hinzugefügt.

`active_battery_fields` erfüllt eine ähnliche Funktion wie `batteries`, aber für Batteriefelder. Ein Batteriefeld ist aktiv, wenn die Ladung der aktiven Batterie des Feldes ungleich 0 ist. Ist ein Batteriefeld nicht aktiv, darf es nicht mehr besucht werden, weil der Roboter dann eine Batterie mit Ladung 0 aufnehmen würde. Das zum aktuellen Batteriefeld (`current_battery.battery_field` oder `new_battery.battery_field`) gehörende Bit aus `active_battery_fields` wird auf 0 gesetzt, wenn `new_battery.charge == 0`. Wie bei `batteries` wird diese Änderung am Ende des Funktionsaufrufs rückgängig gemacht.

`find_way_backtracking` prüft am Anfang mit einem Aufruf von `is_graph_connected`, ob der Graph noch zusammenhängend ist. `is_graph_connected` setzt während der durchgeführten Breitensuche solange Bits der besuchten Batteriefelder in einer Kopie von `active_battery_fields` auf 0, bis keine Bits mehr gesetzt sind – dann wird `True` zurückgegeben – oder die Breitensuche beendet ist. In diesem Fall ist der Graph nicht mehr verbunden, weil nicht alle Batteriefelder besucht werden konnten, und es wird `False` zurückgegeben. Gibt `is_graph_connected` `False` zurück, sucht `find_way_backtracking` nicht weiter nach einem Weg, sondern gibt direkt `None` zurück. Die Breitensuche ist mit `deque` aus dem `collections`-Modul implementiert. Neue Felder werden an die Queue angehängt, und sobald die Queue keine Elemente mehr besitzt, ist die Breitensuche beendet.

Um sich selbst mit einer neuen aktuellen Batterie aufzurufen, iteriert `find_way_backtracking` durch alle Verbindungen der aktuellen Batterie `current_battery.connections`. Die Verbindungen werden vorher in zwei Listen aufgeteilt, `visited` und `unvisited`. `visited` enthält alle Verbindungen, für die `connection.new_battery.battery_field.is_visited`[31] gilt, `unvisited` alle anderen. Wie unter „Lösungsidee“ beschrieben, werden zuerst alle Verbindungen aus `unvisited`, dann alle aus `visited` betrachtet. Eine Verbindung kann betrachtet werden, wenn das neue Batteriefeld noch aktiv ist, wenn also `connection.new_battery.battery_field.active_battery.charge != 0` gilt. Für jede dieser Verbindungen wird `find_way_backtracking` rekursiv aufgerufen, wobei als `new_battery` `connection.new_battery` und als aktuelle Batterie die im Moment aktive Batterie des Batteriefelds übergeben wird. Sobald `find_way_backtracking` einen Weg gefunden hat (`batteries == 0`, s.o.), wird eine Liste mit den besuchten Batteriefeldern und Batterien zurückgegeben, ansonsten `None`. Nach dem rekursiven Aufruf wird daher überprüft, ob der Rückgabewert nicht `None` ist. Ist das der Fall, werden die genutzte Verbindung und die aktuelle Batterie an die Liste vorne angefügt und zurückgegeben.

[30] Dieses Tupel muss nach dem Ändern von `batteries` erstellt werden. Ansonsten ist das Bit für `new_battery` fälschlicherweise nicht gesetzt.
[31] Dieses Attribut wird auf `True` gesetzt, sobald das Batteriefeld besucht wurde. Diese Änderung wird wie üblich am Ende des Funktionsaufrufs rückgängig gemacht.

2.3.1 Generierung des vollständigen Wegs

`Robot.find_way` gibt eine Liste aus Tupeln zurück. Jedes Tupel enthält die besuchte Batterie und die genutzte Verbindung (`BatteryConnection`). Im letzten Tupel steht statt der genutzten Verbindung `None`. Mit diesen Tupeln lässt sich ein vollständiger Weg bestehend aus Positionen generieren. Jede `BatteryConnection` speichert im Attribut `field_connection`, welche Batteriefeldverbindung genutzt wurde. Dadurch kann bei der Generierung des Weges aus einer Batterieverbindung ein Teilweg ermittelt werden. Die Länge des Teilwegs könnte geringer sein als die Ladung, um die der Roboter die Batterie auf dem Weg entladen muss. Diese Differenz ist immer eine gerade Zahl. Die beiden letzten Elemente des Teilwegs werden daher zusätzlich zum Teilweg sooft an den vollständigen Weg angefügt, bis die Differenz ausgeglichen ist. (Dieses Verfahren wird wegen der Teleportationsfelder leicht abgewandelt, siehe Abschnitt 2.1.1.)

2.4 Spielfeldgenerierung

Ein Spielfeld wird von der `generate_stromrallye`-Funktion aus aufgabe1b.py generiert. `generate_stromrallye` erhält als Parameter

- die gewünschte Batterieanzahl `battery_count`,
- eine Funktion `get_random_charge`, die eine neue Ladung zurückgibt,
- ein `random.Random`-Objekt `rand`,
- die Spielfeldgröße `size` (=Breite und Höhe),
- eine Funktion `choose_next_battery`, die eine neue Position aus allen erreichbaren auswählt (ist mit der Funktion `choose_next_battery` vorbelegt), und
- die Wahrscheinlichkeiten, `probability_mark_as_multi_visit`, `probability_do_multi_visit` und `probability_take_fastest`.

Mit der übergebenen Spielfeldgröße wird ein `numpy`-Array `field` erstellt. Im Gegensatz zur Beschreibung in der Lösungsidee ist das Spielfeld nicht unendlich groß, aber mit einer ausreichend großen Spielfeldgröße erreichen die Batterien den Spielfeldrand nicht. `field` enthält Ganzzahlen, die drei Werte annehmen können:

- `0` für ein leeres Feld,
- `1` für ein Feld, das als Weg genutzt wird oder auf dem sich der Roboter zu Beginn befindet, und
- `2` für ein Feld, auf dem sich eine Ersatzbatterie befindet.

Alle Felder in `field`, deren Wert kleiner als 2 ist, können folglich als Wege genutzt werden.

Die Roboterstartbatterie wird mit einer zufällig gewählten Ladung in der Mitte des Spielfelds platziert. Die Ersatzbatterien werden wie in der Lösungsidee beschrieben nacheinander auf das Feld gelegt. Eine neue Batterieposition wird mit der lokalen Funktion `get_next_battery_position` ausgewählt. Diese bestimmt alle Felder, die mit der aktuellen Batterie erreicht werden können, ruft dann mit allen gefundenen Feldern `choose_next_battery` auf und gibt deren Rückgabewert zurück. `choose_next_battery` wählt aus allen möglichen neuen Feldern gemäß der Schwierigkeit und der Anzahl der Felder, die für einen Weg neu als Weg markiert werden müssten, eines aus. `get_next_battery_position` nutzt wieder Maze-Routing, das dem aus Aufgabenteil a) sehr ähnlich ist: Es gibt zwei Listen, `one_visit_positions` und `double_visit_positions`. In diesen beiden Listen werden die Felder gespeichert, die als nächstes mit der nächstniedrigeren Ladung besucht werden sollen. In zwei Schleifen hintereinander wird die noch verbleibende Ladung heruntergezählt. Die erste Schleife zählt bis zu der Ladung, bis zu der Felder doppelt besucht werden müssen (siehe Formel aus der Lösungsidee), die zweite beginnt bei der Ladung, bei der die erste aufgehört hatte, und berücksichtigt nur noch `one_visit_positions`. Wurde eine neue mögliche Batterieposition gefunden, wird diese einem `dict` (`fastest_difficulties2positions` für Positionen, zu denen keine Ladung verfallen gelassen wird, `normal_difficulties2positions` für andere) gespeichert. Die `dicts` haben als

Schlüssel die Schwierigkeit dieser Position (siehe unten) und als Werte eine Liste mit Tupeln. Ein Tupel enthält eine Position und den Weg, mit dem der Roboter zu dieser Position gelangt. `choose_next_battery` wählt aus den Schlüsseln der `dict`s eine Schwierigkeit aus und sucht aus allen Positionen mit dieser Schwierigkeit diejenige aus, mit deren Weg am wenigsten neue Felder in `field` mit 1 markiert werden müssten (d.h. es sollen möglichst wenige Felder für andere Batterien „blockiert“ werden). Gibt es mehrere Positionen mit gleich guten Wegen, wird eine zufällige ausgewählt.

`generate_stromrallye` speichert in einem Array `difficulties`, das dieselbe Größe wie `field` hat, die Schwierigkeiten für jedes Feld ab. Nach dem Initialisieren sind alle Werte im Array auf 1 gesetzt. Für jede erreichbare Position, die von `get_next_battery` gefunden wird, wird die Schwierigkeit auf dem jeweiligen Feld wie beschrieben um 4 erhöht, wenn es ein schnellstes Feld ist, ansonsten um `rand.randint(0, 2)`.

Soll eine Batterie mehrfach besucht werden, wird die Schwierigkeit der zugehörigen Position in `difficulties` auf `255` gesetzt. In `choose_next_battery` wird geprüft, ob ein Schlüssel den Wert `255` hat. Ist dies der Fall, werden die zugeordneten Positionen mit der Wahrscheinlichkeit `PROBABILITY_DO_MULTI_VISIT` ausgewählt. Ansonsten wird aus den verbleibenden Schwierigkeiten wie gewöhnlich eine ausgewählt. `generate_stromrallye` prüft für jede neu ausgewählte Position, ob die Schwierigkeit dieser Position `255` ist. Alle Positionen, die nochmals besucht werden sollen, werden in einem `dict` gespeichert. Aus diesem `dict` erhält die Funktion dann die Batterieladung, mit der sich der Roboter von dieser Position aus weiterbewegen soll. Dies ist die Ladung, um die die Batterieladung, mit der neue Felder gesucht wurden, früher reduziert wurde.

3 Beispiele

3.1 Spielfeldgenerierung

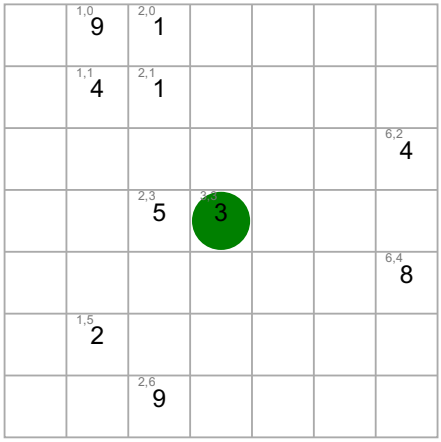
Folgende Spielfelder wurden mit den jeweils angegebenen Parametern generiert. Für alle Beispiele wurde der Startwert 1 an den Konstruktor von `random.Random` übergeben und für folgende Parameter wurden für alle drei Level dieselben Werte gewählt:

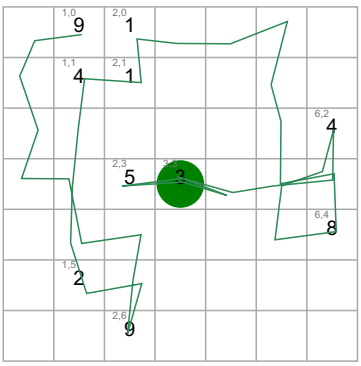
```
FIELD_SIZE = 50   # Größe des 'field'-Arrays
BATTERY_COUNT = 10
NEW_CHARGE = lambda rand: int(round(rand.triangular(0.5, 11.4999, 3)))
PROBABILITY_TAKE_FASTEST = 0.8
```

Für die Level wäre es durchaus sinnvoll, unterschiedliche Werte für `BATTERY_COUNT` zu verwenden. Um die generierten Spielsituationen vergleichbarer zu machen, wurden immer zehn Batterien generiert.

Anfänger*innen

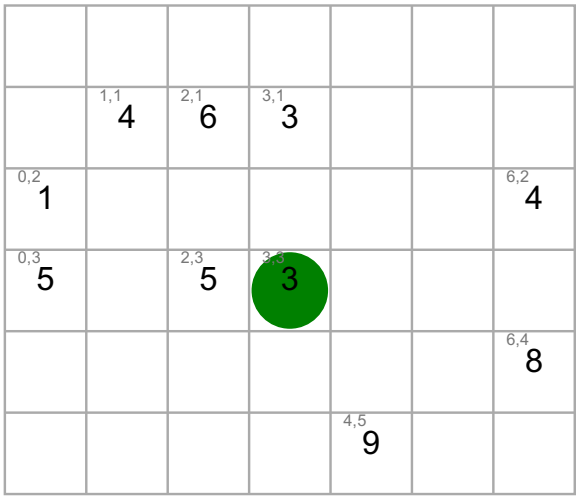
```
PROBABILITY_MARK_AS_MULTI_VISIT = 0.3
PROBABILITY_DO_MULTI_VISIT = 0.6
CALC_FASTEST_WEIGHT = lambda d: d
CALC_NORMAL_WEIGHT  = lambda d: 1
```

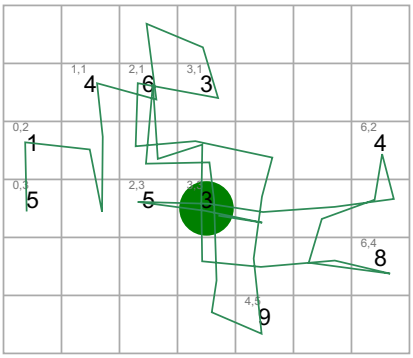




Fortgeschrittene

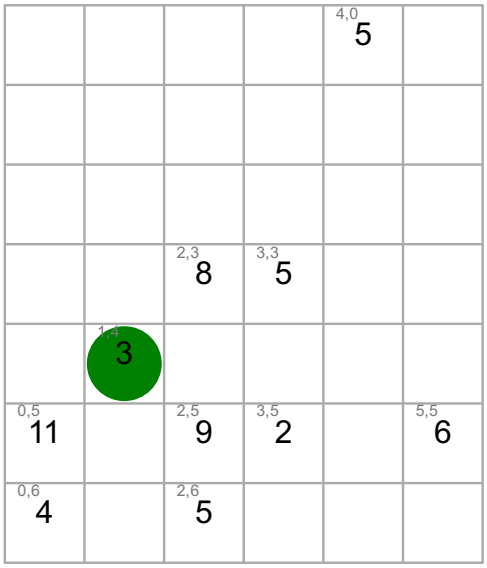
```
PROBABILITY_MARK_AS_MULTI_VISIT = 0.7
PROBABILITY_DO_MULTI_VISIT = 0.6
CALC_FASTEST_WEIGHT = lambda d: d**2
CALC_NORMAL_WEIGHT  = lambda d: d
```

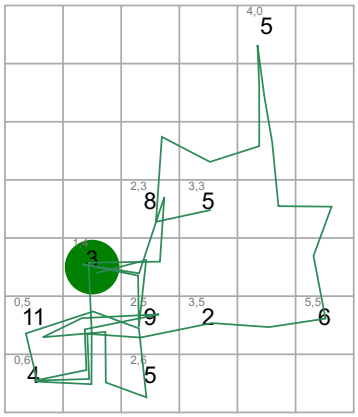




Expertinnen und Experten

```
PROBABILITY_MARK_AS_MULTI_VISIT = 0.9
PROBABILITY_DO_MULTI_VISIT = 0.7
CALC_FASTEST_WEIGHT = lambda d: d**6
CALC_NORMAL_WEIGHT  = lambda d: d*10
```





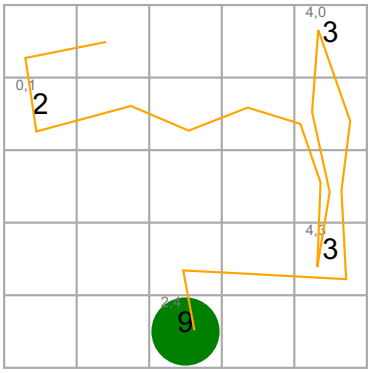
3.2 Beispieldaten

Beim Ausführen von aufgabe1a.py werden die gezeigten Abbildungen im `solutions`-Ordner gespeichert. Die Wege in den Abbildungen werden von `Robot.generate_full_way` erstellt, `save_svg` (aus util.py) speichert das Spielfeld dann als SVG-Datei. Der randomisierte Linienverlauf wurde zum Teil manuell optimiert, damit der Weg besser erkennbar ist.

Die Ausgabe und insbesondere die Zeitangaben stammen von einem Windows-10-Rechner mit Ryzen-5-2600X-Prozessor (3,6 GHz).

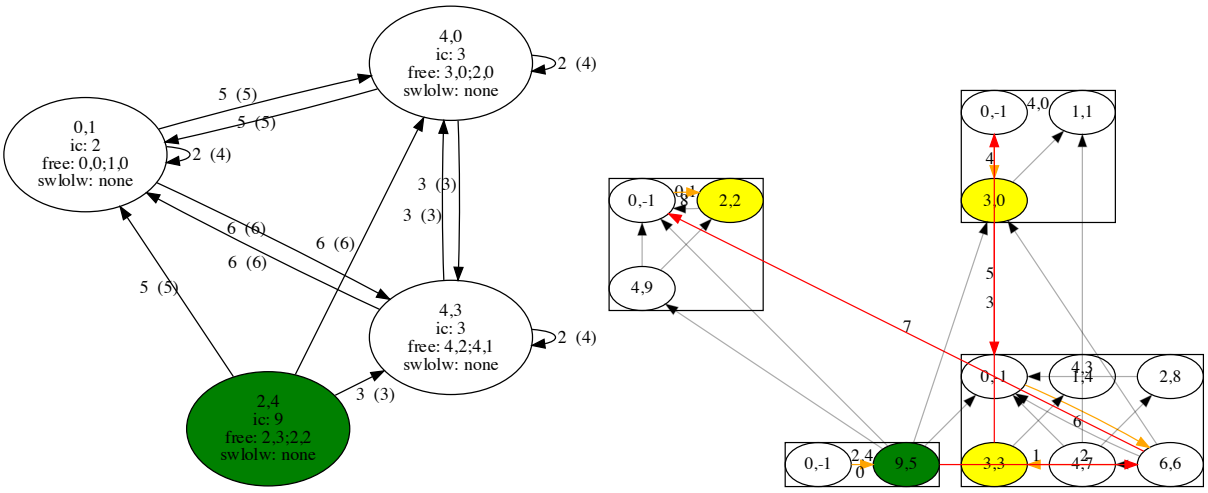
stromrallye0.txt

```
# Ausgabe des Maze-Routings:
searching ways for battery 0 (4, 0)
searching ways for battery 1 (0, 1)
searching ways for battery 2 (4, 3)

# field count: Batteriefeldanzahl (Anzahl Knoten erster Graph)
# battery count: Anzahl Knoten zweiter Graph
field count: 4 battery count: 10

# Backtracking: trying <aktuelle Position> <Ladung von
# current_battery> <Ladung von new_battery>
# Die Einrückung gibt an, auf welcher Rekursionstiefe der
# Aufruf erfolgt.
# Sobald das Feld 0,1 betreten wird, wurde ein Weg gefunden.
# Die Prüfung, ob sich das Feld 0,1 als letztes Batteriefeld
# mit Ladung 2 eignet, wird nicht ausgegeben.
trying (2, 4) 9 0
  trying (4, 3) 3 6
    trying (4, 0) 3 0
      trying (4, 3) 6 0
        trying (0, 1) 2 0

# Anzahl Elemente im Cache; wie oft ein Element im Cache gefunden wurde:
cache size: 5 hits: 0

# Zeit für Backtracking:
time: 0.0002s
start at field (2, 4) with charge 9
go to field (4, 3) and
  drop battery with charge 6 on this field
  grab battery with charge 3
go to field (4, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 3
go to field (4, 3) and
  drop battery with charge 0 on this field
  grab battery with charge 6
go to field (0, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 2
```



Dank der Heuristiken ist der erste Weg, den der Algorithmus versucht, der richtige.

Im Folgenden sind die beiden zugehörigen Graphen dargestellt. Im zweiten Graphen ist die Lösung in rot (Kanten, die den Weg darstellen) und orange (Kanten, die Batterietauschen darstellen) eingezeichnet. Die Kanten für den Weg sind nummeriert.

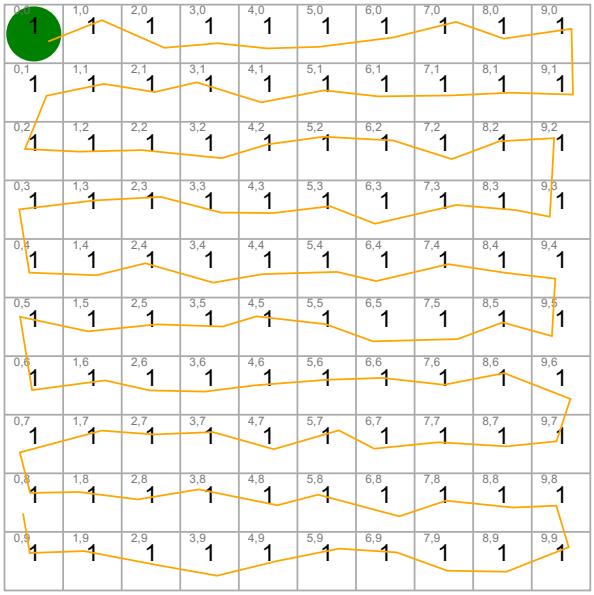


stromrallye1.txt

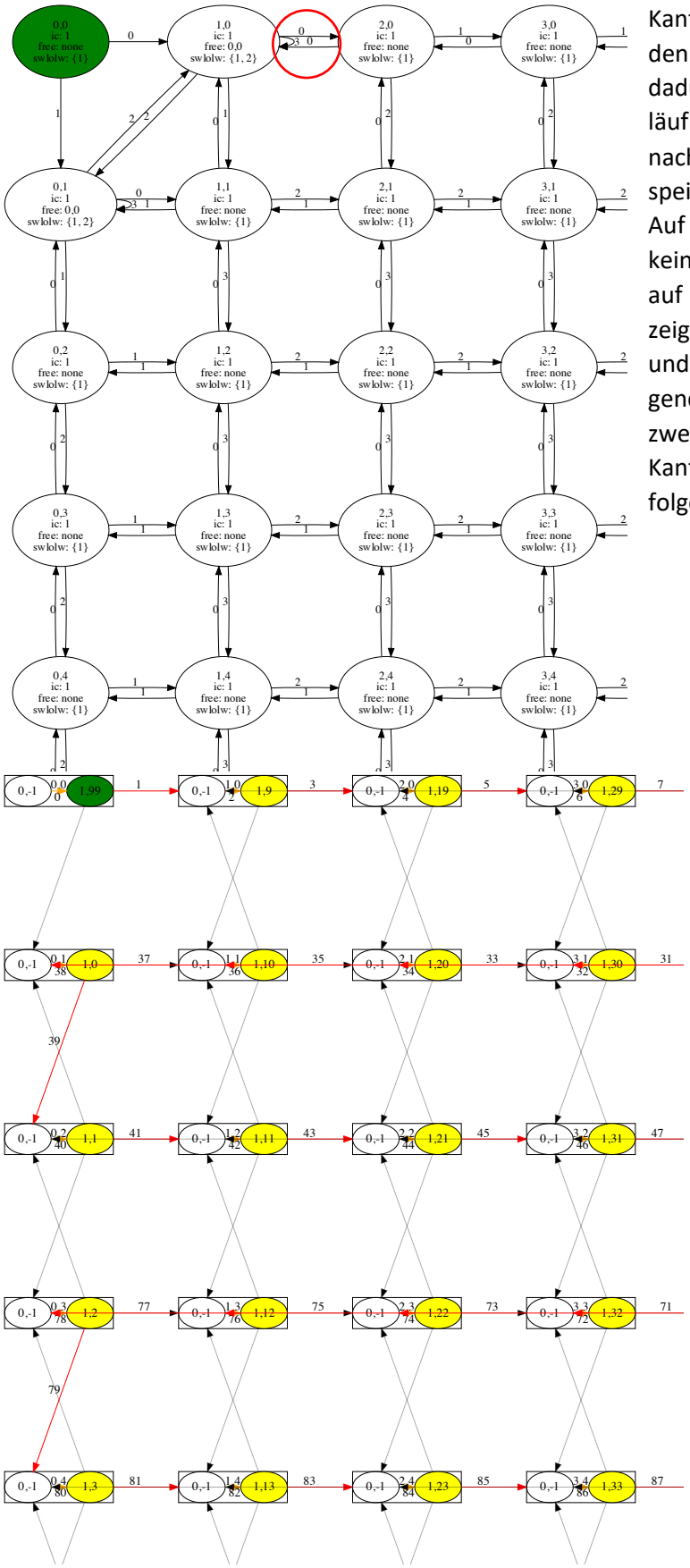
```
searching ways for battery 0 (0, 0)
searching ways for battery 1 (1, 0)
searching ways for battery 2 (2, 0)
[…]
searching ways for battery 98 (8, 9)
field count: 100 battery count: 100
trying (0, 0) 1 0
  trying (1, 0) 1 0
    trying (2, 0) 1 0
      trying (3, 0) 1 0
        trying (4, 0) 1 0
          trying (5, 0) 1 0
            trying (6, 0) 1 0
              trying (7, 0) 1 0
                trying (8, 0) 1 0
                  trying (9, 0) 1 0
                    trying (9, 1) 1 0
                      trying (8, 1) 1 0
                        trying (7, 1) 1 0
                          trying (6, 1) 1 0
                            […]
cache size: 100 hits: 0
time: 0.0079s
```

```
start at field (0, 0) with charge 1
go to field (1, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (2, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (3, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (4, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
[…]
go to field (2, 9) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (1, 9) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (0, 9) and
  drop battery with charge 0 on this field
  grab battery with charge 1
```

Besser geeignet zur Darstellung des Weges ist folgende Abbildung:

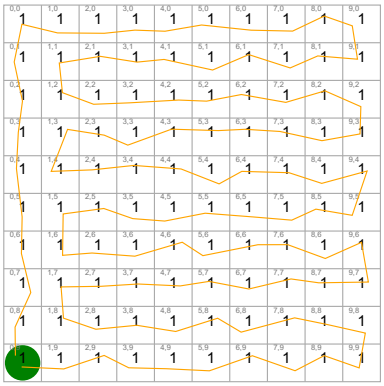


Dieser Weg wird beim ersten Versuch gefunden. Die heuristische Sortierung der Kanten ist für diese Spielsituation nicht wichtig, da alle Kanten ohnehin gleich sind. Ein Weg wird nur so schnell gefunden, weil die Kanten des zweiten Graphen schon sortiert sind. Dabei ist es hilfreich, dass die Batteriefelder zu Beginn nach Position sortiert werden. Mit der Sortierung wird immer zuerst ein Weg nach rechts, dann ein Weg nach unten gesucht (Wege zu Batteriefeldern nach links und oben sind bereits gefunden worden), wodurch die Kanten des ersten Graphen und so auch die des zweiten Graphen sortiert werden. Um dies zu veranschaulichen, wurden im ersten Graphen die Kanten in der Reihenfolge nummeriert, in der sie gespeichert sind. Die Kanten des zweiten Graphen sind in einer ähnlichen Reihenfolge gespeichert; dies ist nicht dargestellt:



Kanten, die mit „0“ beschriftet sind und dennoch nach links zeigen, entstehen dadurch, dass beim Speichern der gegenläufigen Kante (nach rechts zeigend) die nach links zeigende Kante ebenfalls gespeichert wird (siehe bspw. roter Kreis). Auf das Backtracking hat dies jedoch keine Auswirkung, weil das Batteriefeld, auf das die nach links zeigende Kante zeigt, dann ohnehin nicht mehr aktiv ist und so stattdessen die nach rechts zeigende Kante genutzt wird (natürlich im zweiten Graphen und nicht im ersten; die Kanten sind aber in derselben Reihenfolge gespeichert).

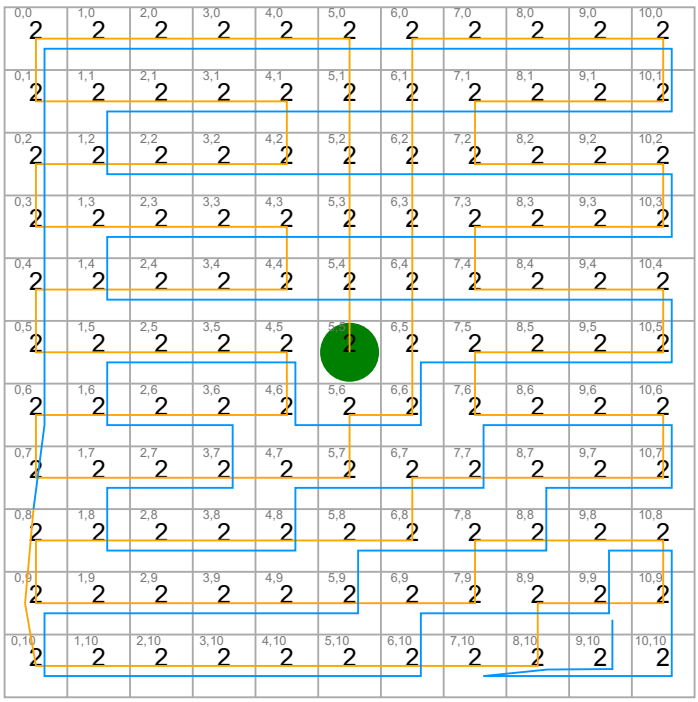
Der Weg wird genauso schnell gefunden, wenn der Roboter in einem der drei anderen Eckfelder startet. Der Weg sieht dann etwas weniger regelmäßig aus (beispieldaten/stromrallye1_unten_links.txt):



stromrallye2.txt

```
searching ways for battery 0 (0, 0)
searching ways for battery 1 (1, 0)
searching ways for battery 2 (2, 0)
[…]
searching ways for battery 118 (8, 10)
searching ways for battery 119 (9, 10)
field count: 121 battery count: 241
trying (5, 5) 2 0
  trying (5, 4) 2 1
    trying (5, 3) 2 1
      trying (5, 2) 2 1
        trying (5, 1) 2 1
          trying (5, 0) 2 1
            trying (4, 0) 2 1
              trying (3, 0) 2 1
                trying (2, 0) 2 1
                  trying (1, 0) 2 1
                    trying (0, 0) 2 1
                      trying (0, 1) 2 1
                        trying (1, 1) 2 1
                          trying (2, 1) 2 1
[…]
cache size: 245 hits: 0
time: 0.0302s
start at field (5, 5) with charge 2
go to field (5, 4) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (5, 3) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (5, 2) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (5, 1) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (5, 0) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (10, 8) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (10, 9) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (10, 10) and
  drop battery with charge 0 on this field
  grab battery with charge 2
go to field (9, 10) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (8, 10) and
  drop battery with charge 1 on this field
  grab battery with charge 1
go to field (7, 10) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (8, 10) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (9, 10) and
  drop battery with charge 0 on this field
  grab battery with charge 1
```

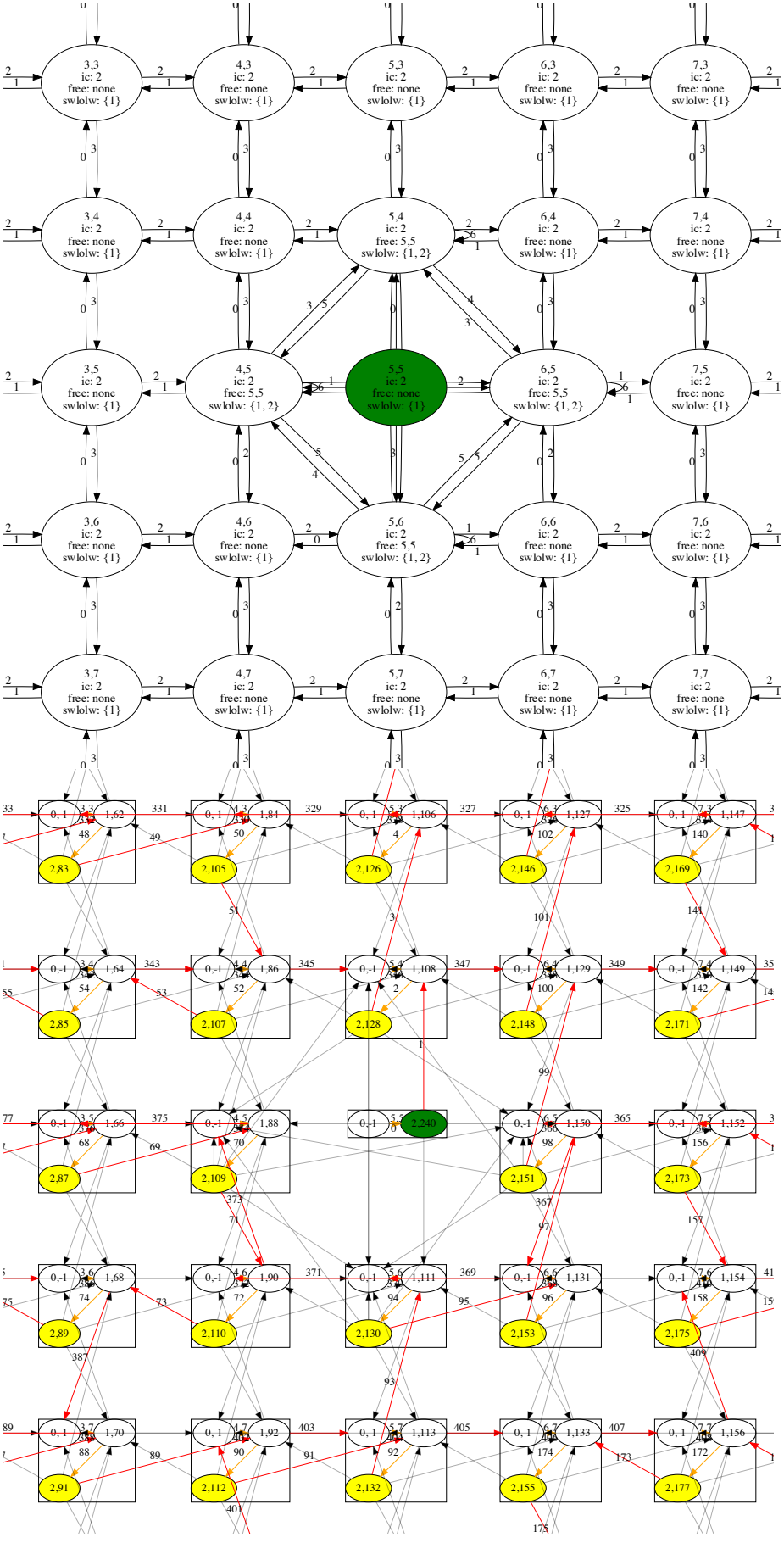
Von den vielen Lösungen für diese Spielsituation ist es am einfachsten, erst alle Batterien einmal zu besuchen, woraufhin alle Batterien nur noch Ladung 1 haben, und dann nochmals alle Batterien zu besuchen. Eine ähnliche Lösung findet der Algorithmus, nur unten rechts werden ein paar Felder am Ende zum ersten Mal besucht. Zur besseren Darstellung wurde die zweite Hälfte des Weges blau eingefärbt:



Die Optimierung, vorzugsweise noch nie besuchte Batteriefelder als nächstes zu besuchen, sorgt dafür, dass durch Backtracking fast beim ersten Versuch ein Weg gefunden wird. Lediglich für das letzte Stück des Weges in der unteren rechten Ecke ist ein zweiter Anlauf nötig:

```
trying (1, 10) 1 0
  trying (2, 10) 1 0
    trying (3, 10) 1 0
      trying (4, 10) 1 0
        trying (5, 10) 1 0
          trying (6, 10) 1 0
            trying (6, 9) 1 0
              trying (7, 9) 1 0
                trying (8, 9) 1 0
                  trying (9, 9) 1 0
                    trying (9, 10) 2 0
                      trying (10, 10) 2 1
                      disconnected
                      trying (8, 10) 1 1
                      disconnected
                    trying (9, 8) 1 0
                      trying (10, 8) 1 0
                        trying (10, 9) 1 0
                          trying (10, 10) 2 0
                            trying (9, 10) 2 1
                              trying (8, 10) 1 1
                                trying (7, 10) 1 0
                                  trying (8, 10) 1 0
                                    trying (9, 10) 1 0
```

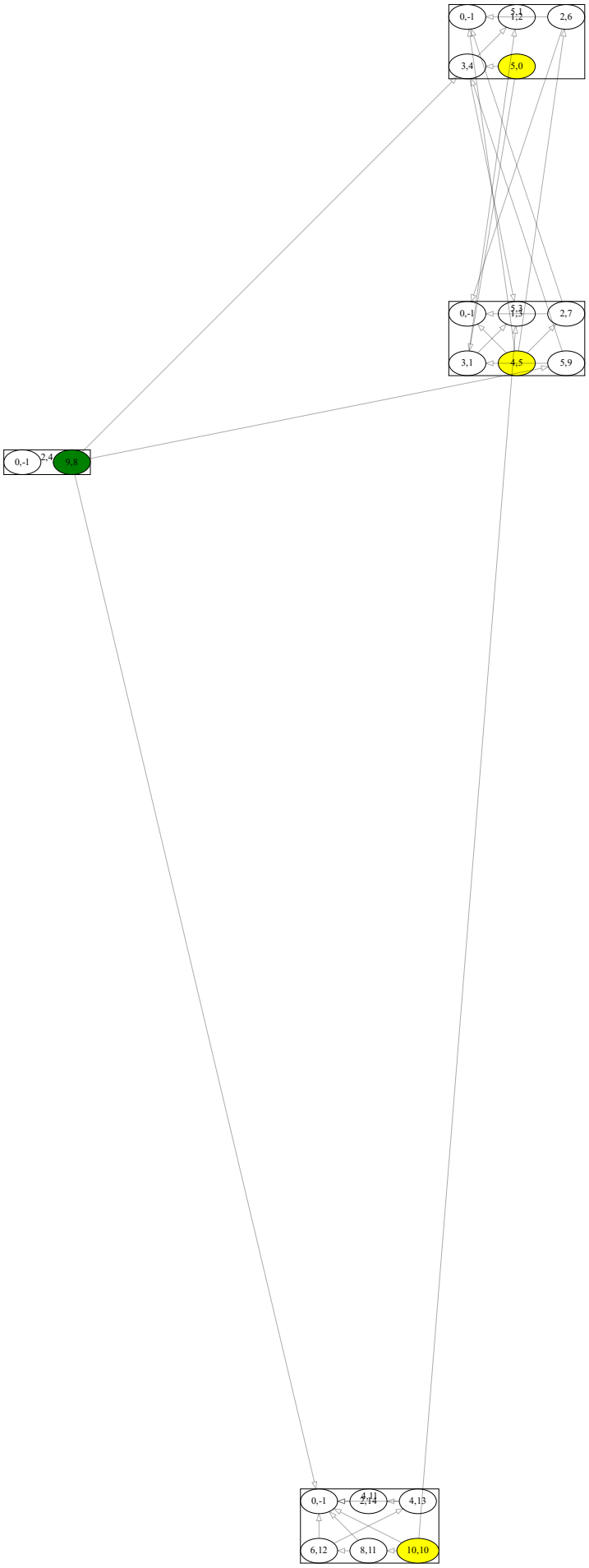
Im Folgenden sind die zugehörigen Graphen dargestellt. Die Kanten des ersten Graphen sind wieder nummeriert:



stromrallye3.txt

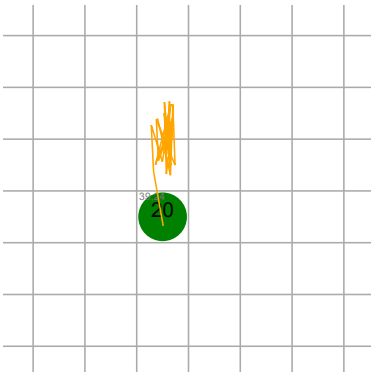
```
searching ways for battery 0 (5, 1)
searching ways for battery 1 (5, 3)
searching ways for battery 2 (2, 4)
field count: 4 battery count: 15
trying (2, 4) 9 0
  trying (5, 3) 4 5
  disconnected
  trying (5, 1) 5 3
  disconnected
  trying (4, 11) 10 0
    trying (5, 3) 4 1
      trying (5, 1) 5 2
        trying (5, 3) 1 3
        disconnected
        trying (5, 1) 2 3
          trying (5, 3) 1 0
          disconnected
          trying (5, 1) 3 0
            trying (5, 3) 1 1
            disconnected
      trying (5, 1) 5 0
        trying (5, 3) 1 3
        disconnected
      trying (5, 3) 1 2
      disconnected
      trying (5, 3) 1 0
      disconnected
cache size: 15 hits: 0
time: 0.0004s
did not find way
```

Für dieses Beispiel gibt es keine Lösung.

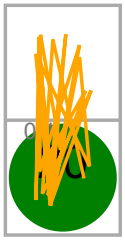


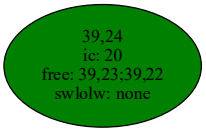
stromrallye4.txt

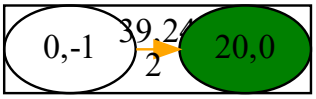
```
field count: 1 battery count: 1
trying (39, 24) 20 0
cache size: 1 hits: 0
time: 0.0001s
start at field (39, 24) with charge 20
```



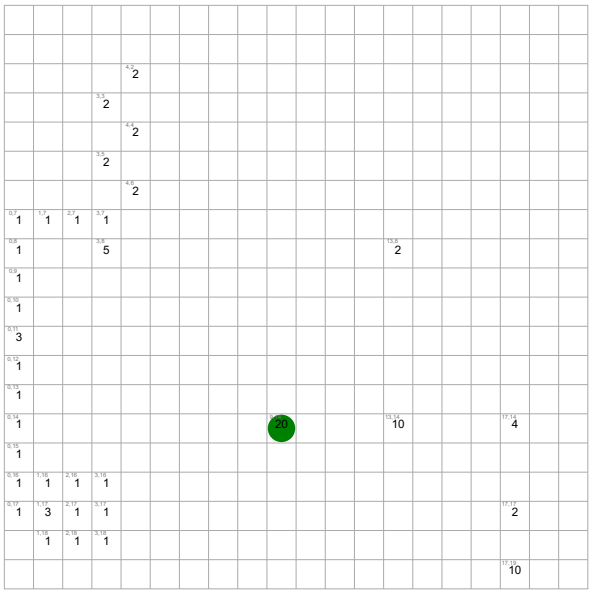
Selbst, wenn das Feld nur zwei Felder groß ist, findet der Algorithmus eine Lösung (beispieldaten/stromrallye4_klein.txt):





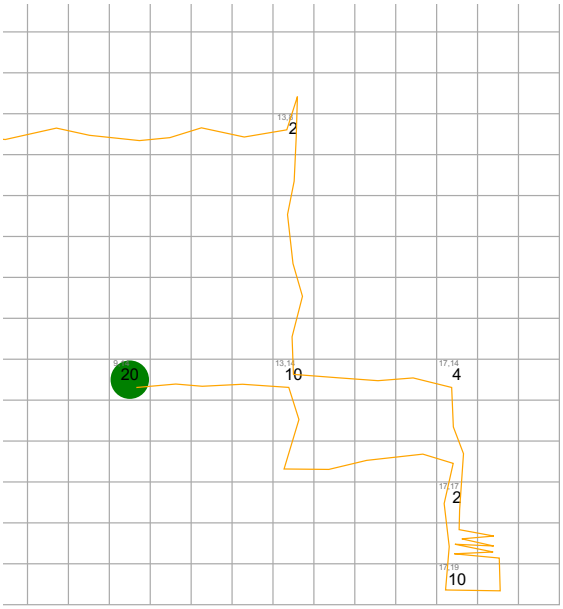


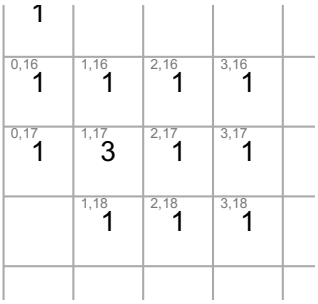
stromrallye5.txt

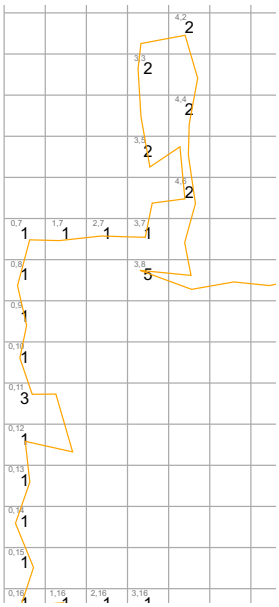


Der Algorithmus benötigt 50 Sekunden, um festzustellen, dass es für stromrallye5.txt keine Lösung gibt. Dass es keine Lösung gibt, lässt sich leicht feststellen: Sobald der Roboter sich zu einer Batterie auf der linken Seite (bis x=4) bewegt, gelangt er nicht mehr zur rechten Seite zurück. Am zweiten Graphen (Datei beispieldaten_svg/stromrallye5_original.txt.gv.svg) lässt sich dies daran erkennen, dass es keine Kante gibt, die von einem Knoten auf der rechten Seite zur linken Seite führt.

Dies bedeutet, dass zuerst alle Batterien auf der rechten Seite leer sein müssen, bevor der Roboter sich zur linken Seite bewegt. Als einzige ist die anfängliche Bordbatterie dazu geeignet, die rechte Seite zu erreichen, und es gibt nur einen Weg, mit dem dies möglich ist (mittlere Abbildung). Auf der linken Seite erreicht der Roboter mit diesem Weg die Batterie mit Ladung 5. Die Aneinanderreihung der Batterien mit Ladung 1 kann nur einmal „durchquert“ werden, in diesem Fall also nach unten. Der Weg des Roboters nach unten ist in der unteren Abbildung dargestellt. Hat der Roboter diesen Weg zurückgelegt, bleiben die Batterien unten links übrig:





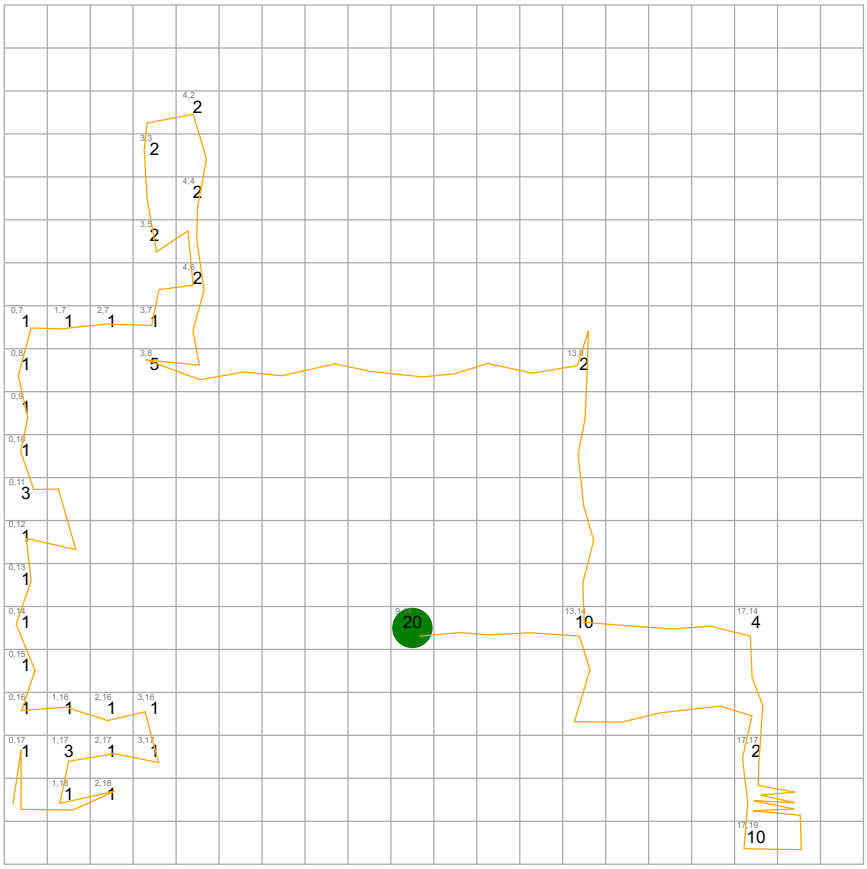


Die Batterien können allerdings nur leer werden, wenn beispielsweise die Batterie auf Feld 3,18 oder auf Feld 0,17 fehlt (zufälligerweise ist die Batterie auf Feld 3,18 diejenige, die in der Datei ganz unten steht und die bis zum 13. Januar durch die zu niedrige Ersatzbatterienanzahl ausgeschlossen war), weil es ansonsten nicht möglich ist, erneut zu dem Feld zu gelangen, auf dem die 3er-Batterie von Feld 1,17 (mit verbleibender Ladung 2) abgelegt wurde und dann die noch nicht entladenen Batterien zu erreichen.

Die Batterie auf Feld 3,18 wurde aus der Beispieldatendatei stromrallye5.txt entfernt. Die originale Datei findet sich unter beispieldaten/stromrallye5_original.txt. Ohne diese Batterie erzeugt das Programm die folgende Ausgabe:

```
searching ways for battery 0 (4, 2)
searching ways for battery 1 (3, 3)
searching ways for battery 2 (4, 4)
searching ways for battery 3 (3, 5)
searching ways for battery 4 (4, 6)
[…]
searching ways for battery 31 (1, 18)
searching ways for battery 32 (2, 18)
field count: 34 battery count: 185
cache size: 147098 hits: 5890
time: 4.5308s
start at field (9, 14) with charge 20
go to field (13, 14) and
  drop battery with charge 16 on this field
  grab battery with charge 10
go to field (17, 17) and
  drop battery with charge 3 on this field
  grab battery with charge 2
go to field (17, 19) and
  drop battery with charge 0 on this field
  grab battery with charge 10
go to field (17, 17) and
  drop battery with charge 0 on this field
  grab battery with charge 3
go to field (17, 14) and
  drop battery with charge 0 on this field
  grab battery with charge 4
go to field (13, 14) and
  drop battery with charge 0 on this field
  grab battery with charge 16
[…]
go to field (2, 16) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (3, 16) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (3, 17) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (2, 17) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (1, 17) and
  drop battery with charge 0 on this field
  grab battery with charge 3
go to field (1, 18) and
  drop battery with charge 2 on this field
  grab battery with charge 1
go to field (2, 18) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (1, 18) and
  drop battery with charge 0 on this field
  grab battery with charge 2
go to field (0, 17) and
  drop battery with charge 0 on this field
  grab battery with charge 1
```

Dass der Cache die Berechnung beschleunigt, wird durch die fettgedruckte Zeile deutlich. Ausgegeben wird, wie viele Elemente der Cache umfasst und wie oft ein Element im Cache gefunden wurde. Ohne den Cache würde das Backtracking 11,5 Sekunden benötigen.

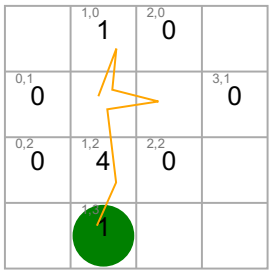


3.3 Sonderfälle

Um zu zeigen, dass der Algorithmus auch die vorgestellten Sonderfälle korrekt löst, sind im Folgenden die Spielbretter der Sonderfälle mit eingezeichneter Lösung dargestellt. Die Dateien für diese Spielsituationen sind auch im `beispieldaten`-Ordner im erweiterten Format enthalten.

Sonderfall aus Abb. 4

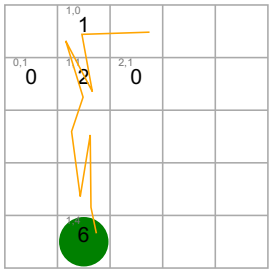
```
start at field (1, 3) with charge 1
go to field (1, 2) and
  drop battery with charge 0 on this field
  grab battery with charge 4
go to field (1, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
```



Dieses Beispiel wurde ein wenig angepasst: Würde der Roboter auf Feld 1,2 starten, so müsste Feld 2,1 nicht leer sein, weil der Roboter auch auf das Roboterstartfeld zurückkehren könnte.

Sonderfall aus Abb. 11

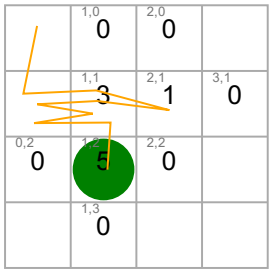
```
start at field (1, 4) with charge 6
go to field (1, 1) and
  drop battery with charge 1 on this field
  grab battery with charge 2
go to field (1, 0) and
  drop battery with charge 1 on this field
  grab battery with charge 1
go to field (1, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (1, 0) and
  drop battery with charge 0 on this field
  grab battery with charge 1
```



Für diesen Sonderfall findet der Algorithmus eine andere Lösung als die im Text zur Abbildung beschriebene.

Sonderfall aus Abb. 13

```
start at field (1, 2) with charge 5
go to field (1, 1) and
  drop battery with charge 4 on this field
  grab battery with charge 3
go to field (1, 1) and
  drop battery with charge 1 on this field
  grab battery with charge 4
go to field (1, 1) and
  drop battery with charge 2 on this field
  grab battery with charge 1
go to field (2, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 1
go to field (1, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 2
```

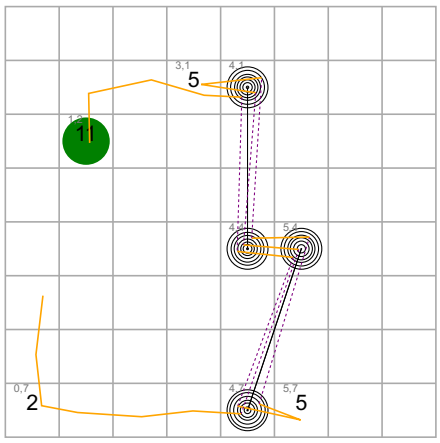


3.4 Teleportationsfelder

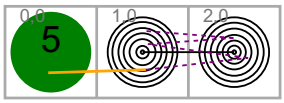
tp0.txt

Der Vollständigkeit halber hier nochmals die Spielsituation mit Teleportationsfeldern aus Abb. 14:

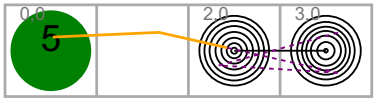
```
field count: 4 battery count: 16
cache size: 5 hits: 0
time: 0.0043s
start at field (1, 2) with charge 11
go to field (3, 1) and
  drop battery with charge 8 on this field
  grab battery with charge 5
go to field (5, 7) and
  drop battery with charge 0 on this field
  grab battery with charge 5
go to field (3, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 8
go to field (0, 7) and
  drop battery with charge 0 on this field
  grab battery with charge 2
```



tp1.txt

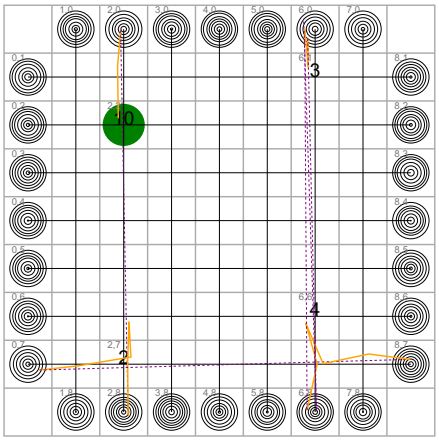


tp2.txt



tp3.txt

```
field count: 4 battery count: 15
cache size: 5 hits: 0
time: 0.0044s
start at field (2, 2) with charge 10
go to field (2, 7) and
  drop battery with charge 6 on this field
  grab battery with charge 2
go to field (2, 7) and
  drop battery with charge 0 on this field
  grab battery with charge 6
go to field (6, 6) and
  drop battery with charge 0 on this field
  grab battery with charge 4
go to field (6, 1) and
  drop battery with charge 0 on this field
  grab battery with charge 3
```



4 Quellcode

Weniger wichtiger Quellcode ist in den folgenden Auszügen nicht enthalten, sondern wird kurz erklärt.

4.1 Ersten Graphen erstellen – Maze-Routing

Die statische Methode `Robot.from_battery_fields` wird von `Robot.from_file` aufgerufen, um aus den `BatteryField`-Instanzen einen Graphen mit Distanzen zu erstellen.

```
@staticmethod
def from_battery_fields(size: Tuple[int, int], robot_battery_field: BatteryField,
                        battery_fields: List[BatteryField]):
# Einrückung entfernt

# Lokale Funktionen:
def create_new_connections(battery_field, other_battery_field, distance, way):
    # Diese Funktion erstellt zwei Verbindungen, zwischen battery_field und other_battery_field
    # 'connections' enthält die beiden Verbindungen (jeweils in eine Richtung)
```

```
    # zwischen den Feldern, in dieser Reihenfolge:
    # [Verbindung von battery_field nach other_battery_field,
    #  Verbindung von other_battery_field nach battery_field]
    # Eine Verbindung kann None sein, wenn das Batteriefeld, auf das die Kante zeigt, das Roboter-
    # startfeld ist, weil es keine Kanten geben sollte, die auf das Roboterstartfeld zeigen.
    connections = []
    if other_battery_field != robot_battery_field:
        connection = BatteryFieldConnection(other_battery_field, distance, way)
        # Attribute 'longer_way' und 'distance_greater_than_two' werden später gesetzt (wenn es
        # überhaupt einen längeren Weg gibt). 'distance_greater_than_two' ist im Moment -1.
        connections.append(connection)
        battery_field.add_connection(connection)
    else:
        connections.append(None)
    if battery_field != robot_battery_field:
        connection = BatteryFieldConnection(battery_field, distance, way[::-1])
        connections.append(connection)
        other_battery_field.add_connection(connection)
    else:
        connections.append(None)
    return connections

def improve_connections(pos, distance_greater_than_two, longer_way):
    # diese Funktion wird aufgerufen, wenn ein längerer Weg als 2 zum Batteriefeld mit Position 'pos'
    # gefunden wurde. Die Verbindungen aus connections_with_small_distances (s.u.) werden aktualisiert.
    connections = connections_with_small_distances[pos]
    if (c := connections[0]) is not None:
        c.longer_way = longer_way
        c.distance_greater_than_two = distance_greater_than_two
    if (c := connections[1]) is not None:
        c.longer_way = longer_way[::-1]
        c.distance_greater_than_two = distance_greater_than_two
    connections_with_small_distances[pos] = None

original_battery_fields.append(robot_battery_field)
battery_fields = original_battery_fields.copy()
battery_fields.extend(tp_fields)
# Batteriefelder, nach ID sortiert (0. Element hat ID 0, 1. Element ID 1 usw.)
all_battery_fields = battery_fields.copy()
battery_fields.sort(key=lambda b: (b.pos.y, b.pos.x))
# IDs zuweisen (Batteriefelder zuerst, dann Teleportationsfelder)
id_ = 0
for id_, battery_field in enumerate(original_battery_fields, id_):
    battery_field.id = id_
robot_battery_field.id = id_
id_ += 1
for id_, teleportation_field in enumerate(tp_fields, id_):
    teleportation_field.id = id_
# Enthält -1 für freie Felder, ansonsten die ID des Batterie- oder Teleportationsfeldes, das sich auf dem
# Feld befindet (ID wird für free_fields_around genutzt, um festzustellen, ob sich ein Teleportationsfeld
# auf einem Feld befindet):
field = np.full((size[1], size[0]), -1, dtype=np.int_)
for battery_field in battery_fields:
    if battery_field != robot_battery_field:
        if field[battery_field.pos] != -1 or battery_field.pos == robot_battery_field.pos:
            raise ValueError(f"There are at least two batteries or teleportation fields at position"
                                                                                "{battery_field.pos}")
        field[battery_field.pos] = battery_field.id

battery_count = len(battery_fields)
for battery_num in range(battery_count - 1):
    battery_field = battery_fields[battery_num]
    if type(other_battery_field) == BatteryField and battery_field.initial_battery.charge == 0:
        # Wenn auf einem Batteriefeld schon zu Anfang eine Batterie mit
        # Ladung 0 liegt, müssen keine Verbindungen gesucht werden
        continue
    # Folgendes Array speichert IDs aller Batterien, für
    # die kein Weg auf einfache Weise gefunden wurde
    batteries_for_maze_routing = np.full(field.shape, -1, dtype=np.int_)
    # Folgendes Array speichert alle Verbindungen, für die
    # per Maze-Routing ein längerer Weg gefunden werden soll
```

```
    connections_with_small_distances = np.empty(field.shape, dtype=list)
    batteries_for_maze_routing_count = 0  # Anzahl der Batterien für Maze-Routing, zu denen ein Weg
                                          # gefunden werden soll
    batteries_with_small_distance_count = 0  # Anzahl der Batterien, zu denen per Maze-Routing
                                             # eine größere Distanz gefunden werden soll
    for other_battery_num in range(battery_num + 1, battery_count):
        assert battery_num != other_battery_num
        other_battery_field = battery_fields[other_battery_num]
        if (type(other_battery_field) == BatteryField and
                                                 other_battery_field.initial_battery.charge == 0):
            continue
        assert battery_field != other_battery_field
        min_x, max_x = sorted((battery_field.pos.x, other_battery_field.pos.x))
        # battery_field.pos.y muss <= other_battery_field.pos.y sein,
        # weil Batteriefelder sortiert wurden
        horizontal = field[battery_field.pos.y, min_x+1:max_x]
        vertical = field[battery_field.pos.y+1:other_battery_field.pos.y, other_battery_field.pos.x]
        if battery_field.pos.y != other_battery_field.pos.y and min_x != max_x:
            corner = field[battery_field.pos.y, other_battery_field.pos.x]
        else:
            # Auf dem Weg biegt der Roboter nicht ab, es gibt kein Eckfeld
            corner = -1
        if np.all(horizontal == -1) and np.all(vertical == -1) and corner == -1:
            distance = (max_x - min_x) + (other_battery_field.pos.y - battery_field.pos.y)

            # Generiere den Weg zwischen battery_field und other_battery_field
            way = [Vector2(x, battery_field.pos.y) for x in
                   (range(battery_field.pos.x+1, other_battery_field.pos.x)
                    if battery_field.pos.x <= other_battery_field.pos.x else
                    range(battery_field.pos.x-1, other_battery_field.pos.x, -1))]  # horizontal
            if battery_field.pos.y != other_battery_field.pos.y and min_x != max_x:
                way.append(Vector2(other_battery_field.pos.x, battery_field.pos.y))  # Eckfeld
            way.extend(Vector2(other_battery_field.pos.x, y) for y in
                       range(battery_field.pos.y+1, other_battery_field.pos.y))  # vertikal
            # Erstelle neue Verbindungen zwischen Batteriefeldern:
            connections = create_new_connections(battery_field, other_battery_field, distance, way)
            if distance > 2:
                for connection in connections:
                    if connection:
                        connection.distance_greater_than_two = distance
            else:
                # Speichere diese beiden Verbindungen für später, es muss ein längerer Weg
                # gefunden werden. Dieser wird im 'longer_way'-Attribut gespeichert werden.
                batteries_with_small_distance_count += 1
                connections_with_small_distances[other_battery_field.pos] = connections
        else:
            # Kein Weg konnte auf einfache Weise gefunden werden,
            # ein Weg muss per Maze-Routing gefunden werden
            batteries_for_maze_routing[other_battery_field.pos] = other_battery_num
            batteries_for_maze_routing_count += 1

    if batteries_for_maze_routing_count != 0 or batteries_with_small_distance_count != 0:
        # MAZE-ROUTING

        # Array 'visited' enthält Mengen mit allen Distanzen (max. 2), die zu diesem Feld
        # gefunden wurden.
        visited = np.frompyfunc(set, 0, 1)(np.empty(field.shape, dtype=object))
        one_visit_positions = {battery_field.pos: []} # {<Position>: <Weg zu Position>}
        double_visit_positions = {}
        for distance in itertools.count(1):  # beginne mit Distanz 1, alle neu gefundenen
                                             # Felder erhalten diese Distanz
            # Folgende dicts speichern die Positionen für den nächsten Schleifendurchlauf.
            # Dadurch, dass die Positionen als Schlüssel in dicts gespeichert werden, wird sichergestellt,
            # dass immer nur eine neue Position gespeichert wird, auch wenn mehrere Wege gefunden werden.
            # Für eine Position ist immer nur ein Weg relevant.
            new_one_visit_positions = {}
            new_double_visit_positions = {}
            do_break = False
            for pos, way in one_visit_positions.items():
                visited[pos].add(distance)
```

```
            # Dem Generatoren Vector2.neighbor_fields werden die Argumente min_x, max_x, min_y, max_y
            # übergeben. Es werden die Positionen aller Nachbarfelder erzeugt, die innerhalb dieser
            # Grenzen liegen.
            for new_pos in pos.neighbor_fields(0,field.shape[1]-1,0,field.shape[0]-1):
                assert 0 <= new_pos.x < field.shape[1] and 0 <= new_pos.y < field.shape[0]
                if connections_with_small_distances[new_pos] is not None:
                    # Auf diesem Feld liegt eine Batterie, zu der eine größere Distanz
                    # gefunden werden soll
                    if distance > 2:
                        # Die aktuelle Distanz ist groß genug; ein längerer Weg wurde
                        # gefunden. Aktualisiere die Verbindungen aus der connections-
                        # Liste entsprechend (mit Aufruf von improve_connections).
                        improve_connections(new_pos, distance, way)
                        batteries_with_small_distance_count -= 1
                        if (batteries_for_maze_routing_count == 0 ==
                                                              batteries_with_small_distance_count):
                            # es gibt keine Felder mehr, für die Maze-Routing durchgeführt werden muss
                            do_break = True
                            break
                if not visited[new_pos]:
                    num = batteries_for_maze_routing[new_pos]
                    if num != -1:
                        # Eine Batterie, für die ein Weg gefunden werden soll,
                        # befindet sich auf diesem Feld
                        other_battery_field = battery_fields[num]
                        assert battery_field != other_battery_field
                        batteries_for_maze_routing_count -= 1
                        batteries_for_maze_routing[new_pos] = -1
                        connections = create_new_connections(battery_field, other_battery_field,
                                                             distance, way)
                        if distance > 2:
                            for connection in connections:
                                if connection:
                                    connection.distance_greater_than_two = distance
                        else:
                            batteries_with_small_distance_count += 1
                            connections_with_small_distances[other_battery_field.pos] = connections

                        if batteries_for_maze_routing_count == 0 == \
                                                              batteries_with_small_distance_count:
                            do_break = True
                            break
                    # Im Folgenden wird statt 'if' nicht 'elif' verwendet, weil auf new_pos auch die
                    # Roboterbatterie liegen könnte. In diesem Fall kann es sein, dass ein Weg per Maze-
                    # Routing gesucht werden soll,aber das Roboterstartfeld kann trotzdem betreten werden.
                    if field[new_pos] == -1:
                        # Dieses Feld kann als nächstes besucht werden
                        new_one_visit_positions[new_pos] = way + [new_pos]
                else:
                    # Dieses Feld wurde bereits einmal besucht, aber es darf nochmals besucht werden
                    if field[new_pos] == -1:
                        new_double_visit_positions[new_pos] = way + [new_pos]
            if do_break:
                break
        if do_break:
            # Statt zum Beenden des Maze-Routings die Variable do_break zu benutzen, wäre es in
            # Python auch möglich, unübersichtliche for-else Konstrukte mit continue-Anweisungen zu
            # benutzen - besser geeignet ist diese Variante mit do_break.
            break
        if batteries_with_small_distance_count != 0 and distance < 5:
            # Es gibt immer noch Batterien, zu denen ein längerer Weg gesucht wird und
            # mit der aktuellen Distanz ist es noch sinnvoll, Felder doppelt zu besuchen
            for pos, way in double_visit_positions.items():
                visited[pos].add(distance)
                for new_pos in pos.neighbor_fields(0, field.shape[1]-1, 0, field.shape[0]-1):
                    if distance not in visited[new_pos]:
                        if connections_with_small_distances[new_pos]:
                            if distance > 2:
                                improve_connections(new_pos, distance, way)
                                batteries_with_small_distance_count -= 1
```

```
                                if (batteries_for_maze_routing_count == 0 ==
                                                                    batteries_with_small_distance_count):
                                    do_break = True
                                    break
                            if field[new_pos] == 0:
                                new_double_visit_positions[new_pos] = way + [new_pos]
                    if do_break:
                        break
            if (do_break or
                    # Falls es keine neuen Felder mehr gibt, wird auch abgebrochen:
                    (not new_one_visit_positions and
                    # new_double_visit_positions wird auch befüllt, wenn distance > 5;
                    # dann ist dieses dict aber nicht mehr relevant
                     (distance > 5 or not new_double_visit_positions))):
                break
            one_visit_positions = new_one_visit_positions
            double_visit_positions = new_double_visit_positions

# lokale Funktion für Dijkstra-ähnlichen Algorithmus:
def get_teleportation_reachable_battery_fields(start_field: BatteryField):
    # battery_fields enthält auch TeleportationFields:
    even_distances = {battery_or_tp_field: (math.inf, None) for battery_or_tp_field in battery_fields}
    odd_distances = even_distances.copy()
    # es muss kein Weg mit gerader Distanz zu start_field gefunden
    # werden, einen solchen gibt es sowieso (Schleife)
    del even_distances[start_field]
    # Initialisiere die von start_field erreichbaren Teleportationsfelder mit Distanz
    for connection in start_field.tp_connections:
        # connection.other_field ist ein Teleportationsfeld, die Distanz des mit diesem verbundenen
        # Tp-Feldes wird gesetzt.
        # +1, weil Teleportation die Ladung der Bordbatterie um 1 reduziert:
        distance_to_neighbor = connection.distance + 1
        neighbor = connection.other_field.paired_field
        way_to_neighbor = connection.way + [connection.other_field.pos, "tp"]
        if distance_to_neighbor % 2 == 0:
            if neighbor in even_distances and distance_to_neighbor < even_distances[neighbor][0]:
                even_distances[neighbor] = (distance_to_neighbor, way_to_neighbor)
        else:
            if neighbor in odd_distances and distance_to_neighbor < odd_distances[neighbor][0]:
                odd_distances[neighbor] = (distance_to_neighbor, way_to_neighbor)
    while even_distances or odd_distances:
        # Suche den Knoten, der als nächstes besucht werden soll (mit kleinster Distanz)
        if even_distances:
            current_even_node, (even_distance, even_way) = min(even_distances.items(),
                                                               key=lambda n: n[1][0])
        else:
            even_distance = math.inf
        if odd_distances:
            current_odd_node, (odd_distance, odd_way) = min(odd_distances.items(),
                                                            key=lambda n: n[1][0])
        else:
            odd_distance = math.inf
        if even_distance == math.inf == odd_distance:
            # Es gibt nur noch Knoten, die nicht erreicht werden können
            break
        if even_distance < odd_distance:
            # current_even_node mit Distanz even_distance wird besucht
            current_node = current_even_node
            distance = even_distance
            way = even_way
            nodes = even_distances
        else:
            current_node = current_odd_node
            distance = odd_distance
            way = odd_way
            nodes = odd_distances
        del nodes[current_node]
        if way and type(current_node) == BatteryField:
            # Ein Weg zu einem Batteriefeld, der Teleportationsfelder nutzt, wurde gefunden
            yield current_node, way, distance
        else:
```

```
            # current_node ist ein TeleportationField, Nachbarknoten werden aktualisiert
            for connection in itertools.chain(current_node.connections,current_node.tp_connections):
                distance_to_neighbor = distance + connection.distance
                neighbor = connection.other_field
                way_to_neighbor = way + [current_node.pos] + connection.way
                if distance_to_neighbor % 2 == 0:
                    if (neighbor in even_distances and
                            distance_to_neighbor < even_distances[neighbor][0]):
                        even_distances[neighbor] = (distance_to_neighbor, way_to_neighbor)
                else:
                    if (neighbor in odd_distances and
                            distance_to_neighbor < odd_distances[neighbor][0]):
                        odd_distances[neighbor] = (distance_to_neighbor, way_to_neighbor)

def get_free_neighbor_fields(pos: Vector2):
    # Generator, der freie Nachbarfelder (auch Teleportationsfelder) erzeugt. Von freien Feldern wird die
    # Position zurückgegeben, für Teleportationsfelder ein TeleportationField-Objekt.
    for p in pos.neighbor_fields(0, field.shape[1] - 1, 0, field.shape[0] - 1):
        if field[p] == -1:
            yield p
        elif type(f := all_battery_fields[field[p]]) == TeleportationField:
            yield f

# Füge Kante mit Distanz 1 zum jeweils verbundenen Teleportationsfeld hinzu
for tp_field in tp_fields:
    tp_field.add_connection(BatteryFieldConnection(tp_field.paired_field, 1, ["tp"]))

for battery_field in original_battery_fields:  # original_battery_fields enthält nur Batteriefelder
    # Speichere Weglängen kleiner als 2 ohne längere Wege ("swlow")
    for connection in battery_field.connections:
        if connection.distance < 3 and connection.distance_greater_than_two not in (3, 4):
            battery_field.small_way_lengths_without_longer_ways.add(connection.distance)

    # Suche zusammenhängende freie Felder
    for free_pos1 in get_free_neighbor_fields(battery_field.pos):
        # Zumindest ist ein Nachbarfeld frei, vielleicht hat dieses Nachbarfeld auch ein freies
        # Nachbarfeld
        if type(free_pos1) == TeleportationField:
            # Es gibt ein Nachbarfeld, das ein Teleportationsfeld ist (siehe Abschnitt 2.1.1)
            battery_field.free_fields_around = [free_pos1, None]
            break
        battery_field.free_fields_around = [free_pos1]  # at least one free field exists
        for free_pos2 in get_free_neighbor_fields(free_pos1):
            # Auch das freie Nachbarfeld hat ein freies Nachbarfeld, d.h. mit jeder beliebigen Ladung
            # kann battery_field als letztes besucht werden.
            if type(free_pos2) == TeleportationField:
                # Ein Teleportationsfeld ist ein Nachbarfeld eines freien Nachbarfeldes
                battery_field.free_fields_around = [free_pos1, free_pos2]
                break
            # Es gibt zwei zusammenhängende freie Felder:
            battery_field.free_fields_around = [free_pos1, free_pos2]
            break
        else:
            # die Schleife ist nicht per break beendet worden, versuche auch das nächste freie
            # Nachbarfeld,vielleicht hat dieses Nachbarfeld freie Nachbarfelder
            continue
        break

    # Füge eine Verbindung von battery_field nach battery_field hinzu
    if battery_field.initial_battery.charge != 0 and battery_field != robot_battery_field:
        if len(battery_field.free_fields_around) == 2:
            # Mit zwei freien Feldern kann jede beliebige (gerade) Ladung auf dem Weg verfallen
            # gelassen werden
            battery_field.add_connection(BatteryFieldConnection(
                battery_field,
                # normaler Weg (distance, way)
                2, [battery_field.free_fields_around[0]],
                # längerer Weg (distance_greater_than_two, longer_way)
                4, [battery_field.free_fields_around[i] for i in (0, 1, 0)])
            )
```

```
        elif len(battery_field.free_fields_around) == 1:
            # Es ist nur ein Weg der Länge 2 möglich
            battery_field.add_connection(
                      BatteryFieldConnection(battery_field, 2, [battery_field.free_fields_around[0]]))

    # Füge Kanten hinzu, deren Wege Teleportationsfelder nutzen
    for other_battery_field, way, distance in get_teleportation_reachable_battery_fields(battery_field):
        # way wird in ein dreielementiges Tupel aufgeteilt, wie beschrieben
        tp_index = way.index("tp")
        # die Folgenden zwei Felder können (mit "tp" dazwischen) beliebig oft wiederholt werden, um
        # Ladung verfallen zu lassen
        first_tp_field_pos = way[tp_index-1]
        second_tp_field_pos = way[tp_index+1]
        way = (way[:tp_index+1],
               [second_tp_field_pos, "tp", first_tp_field_pos, "tp"],
               way[tp_index+1:])
        battery_field.add_connection(BatteryFieldConnection(other_battery_field, distance, way,
                                                            distance, None))
    # Cleanup: Entferne überflüssige Kanten
    battery_field.connections.sort(key=lambda b: b.distance)
    # Gehe alle Kanten mit kleiner werdender Distanz durch
    for i in range(len(battery_field.connections)-1, -1, -1):
        connection = battery_field.connections[i]
        # Gehe alle Kanten durch, die 'connection' überflüssig machen könnten
        for other_connection in battery_field.connections[:i]:
            if other_connection.other_field == connection.other_field:
                # Beide Kanten müssen auf dasselbe Feld zeigen
                assert other_connection != connection
                assert other_connection.distance <= connection.distance
                # Prüfe, ob connection durch other_connection überflüssig wird
                if (connection.distance % 2 == other_connection.distance % 2 and
                        # Entweder muss der längere Weg von other_connection kleiner sein, ...
                        ((connection.distance_greater_than_two >= other_connection.distance and
                          other_connection.distance != -1) or
                        # ... oder beide Kanten haben keinen längeren Weg
                         connection.distance_greater_than_two == -1 == other_connection.distance)):
                    # Die Kante ist überflüssig
                    del battery_field.connections[i]
                    break
return Robot(original_battery_fields, robot_battery_field, size)
```

4.2 Zweiten Graphen erstellen – Fakebatterien

Methode `Robot.create_fake_batteries`:

```
def create_fake_batteries(self):
# Einrückung entfernt
battery_id = 0  # Zähler für Batterie-IDs
# die Verbindungen jedes Batteriefelds wurden bereits von from_battery_fields beim Entfernen
# überflüssiger Kanten nach Distanz sortiert

def search_connections(current_battery: Battery):
    def add_neighbor(charge, field_connection: BatteryFieldConnection, uses_long_way):
        nonlocal battery_id
        # Füge current_battery eine Nachbarbatterie mit Ladung charge auf Feld
        # field_connection.other_field hinzu
        if charge == 0:
            # Batterie mit Ladung 0 benötigt keine ID und es müssen
            # keine Nachbarn für diese Batterie gesucht werden
            current_battery.connections.append(
              BatteryConnection(field_connection.other_field.get_or_create_fake_battery(0)[1],
                                field_connection, uses_long_way))
        else:
            is_new, new_battery = field_connection.other_field.get_or_create_fake_battery(charge)
            if is_new:
                # die Batterie ist neu, daher suche nach Verbindungen für diese Batterie und
                # weise ihr eine neue ID zu
                new_battery.id = battery_id
                battery_id += 1
                search_connections(new_battery)
            current_battery.connections.append(
```

```
                BatteryConnection(new_battery, field_connection, uses_long_way))

    for field_connection in current_battery.battery_field.connections:
        if field_connection.distance > current_battery.charge:
            # BatteryConnections sind nach Distanz sortiert, daher sind auch die Distanzen
            # aller folgenden BatteryConnections zu groß
            break
        if field_connection.distance == current_battery.charge:
            add_neighbor(0, field_connection, False)
        else:
            assert field_connection.distance < current_battery.charge
            if (field_connection.distance % 2 == current_battery.charge % 2 and
                    (current_battery.charge < 3 or
                     # Wenn current_battery.charge > 2, dann muss es einen längeren Weg geben und
                     # dessen Länge muss kleiner als oder gleich current_battery.charge sein
                     (field_connection.distance_greater_than_two != -1 and
                      field_connection.distance_greater_than_two <= current_battery.charge))
            ):
                # Eine Batterie mit Ladung 0 kann auf field_connection.other_field gelegt
                # werden
                add_neighbor(0, field_connection,
                             # ob der längere Weg genutzt wird:
                             current_battery.charge > 2 and field_connection.distance < 3)
            # Die Ladung einer Batterie, die der Roboter auf field_connection.other_field
            # maximal ablegen kann, wenn er mit current_battery startet:
            maximum_remaining_charge = current_battery.charge - field_connection.distance
            # Es kann also eine Batterie mit Ladung maximum_remaining_charge auf diesem Feld geben
            add_neighbor(maximum_remaining_charge, field_connection, False)
            # Zusätzlich kann es Fakebatterien für jede "small way length without longer way" geben
            for small_distance in field_connection.other_field.small_way_lengths_without_longer_ways:
                # small_distance kann 1 oder 2 sein
                if small_distance != maximum_remaining_charge:
                    if maximum_remaining_charge % 2 == small_distance % 2:
                        distance = current_battery.charge - small_distance
                        # Gleiche Bedingung wie oben (distance statt battery.charge):
                        if (distance < 3 or
                                (field_connection.distance_greater_than_two != -1 and
                                 field_connection.distance_greater_than_two <= distance)):
                            add_neighbor(small_distance, field_connection,
                                         distance > 2 and field_connection.distance < 3)

for battery_field in self.battery_fields:
    is_new, battery = battery_field.get_or_create_fake_battery(battery_field.initial_battery.charge)
    # Die Ersatzbatterie kann auch eine Fakebatterie sein und daher kann es sein, dass
    # Verbindungen für diese Batterie bereits zuvor durch einen rekursiven Aufruf berechnet
    # wurden.
    if is_new:
        battery.id = battery_id
        battery_id += 1
        search_connections(battery)

# Sortiere alle Batterieverbindungen (Heuristik zur Optimierung)
for battery_field in self.battery_fields:
    for battery in battery_field.batteries.values():
        battery.connections.sort(
            key=lambda c: (
                # Verbindungen, die zu...
                #  1. ... Batterien führen, die sich auf einem anderen Batteriefeld befinden, zuerst
                0 if c.new_battery.battery_field != battery.battery_field else 1,
                #  2. ... geladenen Batterien führen, die keine Verbindungen haben, zuletzt
                0 if c.new_battery.connections or c.new_battery.charge == 0 else 1,
                #  3. ... Batterien mit höherer Ladung führen zuerst (es kann sinnvoll sein, das '-'
                #         wegzunehmen und dadurch zuerst Batterien mit niedriger Ladung zu betrachten)
                -c.new_battery.charge,
                #  4. ... weiter entfernten Batteriefeldern führen zuerst
                -c.field_connection.distance
            )
        )
        battery_field.neighbors.update(c.new_battery.battery_field for c in battery.connections)
```

```
if 0 not in self.robot_battery_field.batteries:
    # In diesem Fall ist self.robot_battery_field.get_active_battery().charge ungerade und
    # deshalb wurde keine Batterie mit Ladung 0 auf dem Roboterstartfeld erstellt, das
    # Roboterstartfeld benötigt allerdings eine leere Batterie.
    self.robot_battery_field.get_or_create_fake_battery(0)
```

4.3 Backtracking

```
def find_way(self):
# Einrückung entfernt
def is_graph_connected(current_battery: Battery):
    # Diese Breitensuche sollte selbsterklärend sein
    fields = deque(c.new_battery.battery_field for c in current_battery.connections)
    unvisited_fields = active_battery_fields  # nicht die Bits in active_battery_fields direkt ändern
    if unvisited_fields == 0:
        return True
    while fields:
        field = fields.pop()
        if unvisited_fields & (mask := 1 << field.id):
            unvisited_fields &= ~mask
            if unvisited_fields == 0:
                return True
            fields.extendleft(field.neighbors)
    return False

cache = set()
cache_hits = 0

def find_way_backtracking(current_battery: Battery, new_battery: Battery):
    # Code für Ausgabe wurde entfernt
    nonlocal active_battery_fields, batteries  # s.u.
    nonlocal cache_hits
    # Markiere current_battery als inaktiv
    batteries &= ~(1 << current_battery.id)

    if new_battery.charge != 0:
        # markiere Batterie mit ID new_battery.id als aktiv
        batteries |= 1 << new_battery.id
    else:
        # markiere Batteriefeld mit ID current_battery.battery_fild.id als inaktiv
        active_battery_fields &= ~(1 << current_battery.battery_field.id)

    key = (batteries, current_battery)
    if key in cache:
        # Es kann kein Weg gefunden werden
        cache_hits += 1
        # Mache Veränderungen an active_battery_fields und batteries rückgängig
        active_battery_fields |= 1 << current_battery.battery_field.id
        if new_battery.charge != 0:
            batteries &= ~(1 << new_battery.id)
        batteries |= 1 << current_battery.id
        return None
    cache.add(key)

    if batteries == 0:
        # Prüfe, ob der Roboter die Ladung von current_battery verfallen lassen kann
        if (current_battery.charge <= 1 or
                (current_battery.charge == 2 and
                 len(current_battery.battery_field.free_fields_around) >= 1) or
                len(current_battery.battery_field.free_fields_around) == 2):
            # alle Batterien sind leer, es wurde ein Weg gefunden
            return [(current_battery,
                     None)]  # Es gibt kein Batteriefeld, das als nächstes betreten wird
        return None

    # Überprüfe, dass der Graph noch zusammenhängend ist
    if not is_graph_connected(current_battery):
        # Der Graph ist nicht zusammenhängend, es kann kein Weg gefunden werden
        # Mache Veränderungen an active_battery_fields und batteries rückgängig
        active_battery_fields |= 1 << current_battery.battery_field.id
        if new_battery.charge != 0:
```

```
            batteries &= ~(1 << new_battery.id)
        batteries |= 1 << current_battery.id
        return None

    # Ändere aktive Batterie des Batteriefelds
    current_battery.battery_field.active_battery = new_battery
    old_is_visited = current_battery.battery_field.is_visited
    current_battery.battery_field.is_visited = True

    # Teile current_battery.connections auf, je nachdem, ob other_field
    # bereits besucht wurde oder nicht (Heuristik zur Optimierung)
    unvisited = []
    visited = []
    for connection in current_battery.connections:
        if connection.new_battery.battery_field.is_visited:
            visited.append(connection)
        else:
            unvisited.append(connection)
    for connection in itertools.chain(unvisited, visited):
        other_field = connection.new_battery.battery_field
        if other_field.active_battery.charge != 0:
            way = find_way_backtracking(other_field.active_battery, connection.new_battery)
            if way is not None:
                return [(current_battery, connection)] + way

    # Mache die Veränderungen rückgängig
    current_battery.battery_field.active_battery = current_battery
    current_battery.battery_field.is_visited = old_is_visited
    active_battery_fields |= 1 << current_battery.battery_field.id
    if new_battery.charge != 0:
        batteries &= ~(1 << new_battery.id)
    batteries |= 1 << current_battery.id
    return None

robot_battery = self.robot_battery_field.active_battery
if robot_battery.charge == 0:
    return None
active_battery_fields = 0
batteries = 0
for field in self.battery_fields:
    if field.initial_battery.charge != 0:
        active_battery_fields |= 1 << field.id
        batteries |= 1 << field.initial_battery.id

res = find_way_backtracking(robot_battery, self.robot_battery_field.batteries[0])
print("cache size:", len(cache), "hits:", cache_hits)
return res
```

4.4 Generierung des vollständigen Weges

Der vollständige Weg, also eine Liste aus Positionen, wird von `Robot.generate_full_way` aus dem Rückgabewert von `find_way` erstellt. Die Methode ist ab Zeile 820 in aufgabe1a.py zu finden.

4.5 Spielfeldgenerierung

Der wichtigste Quellcode aus aufgabe1b.py:

```
class NoNewPositionsError(IndexError):
    """
    Wird geworfen, wenn der Roboter sich nicht mehr bewegen kann, wenn der Roboter mit der
    aktuellen Batterie also auf keine Felder mehr gelangt.
    """
    pass

def generate_stromrallye(size: int, battery_count, rand: Random,
                         get_random_charge: Callable[[Random], int],
                         probability_mark_as_multi_visit: float,
                         probability_do_multi_visit: float,
                         probability_take_fastest: float,
                         choose_next_battery: Optional[Callable]
```

```
                                        = choose_next_battery):  # choose_next_battery: s.u.
def get_next_battery_position(current_pos: Vector2, initial_charge: int):
    """
    Suche neue Batteriepositionen und gib eine davon zurück (ausgewählt mit choose_next_battery)
    :param current_pos: Position der alten Batterie
    :param initial_charge: Ladung, die sich auf initial_charge befindet
    :return: (Position, Weg zu dieser Position)
    """
    # Diese zwei dicts ordnen lokale Schwierigkeitsgrade Positionen (inkl. Wege) zu. Ein Weg ist eine
    # Liste aus Positionen.
    fastest_difficulties2positions = {}  # schnellste Felder
    normal_difficulties2positions = {}  # normale Felder

    def add_new_position(pos, way, is_fastest):
        # Diese Funktion wird für eine mögliche neue Position (pos) aufgerufen, zu der der Roboter
        # durch den Weg way gelangen kann. is_fastest ist True, wenn pos ein schnellstes Feld darstellt.
        if field[pos] == 0 or difficulties[pos] == 255:  # Schwierigkeitsgrad == 255 heißt, dass
                                                         # das Feld mehrfach besucht werden soll
            difficulty = difficulties[pos]  # Schwierigkeitsgrad für pos
            if difficulties[pos] != 255:
                # Erhöhe Schwierigkeitsgrad an Position pos
                difficulties[pos] += 4 if is_fastest else rand.randint(0, 2)
            difficulties2positions = (fastest_difficulties2positions if is_fastest else
                                      normal_difficulties2positions)
            try:
                difficulties2positions[difficulty].append((pos, way))
            except KeyError:
                difficulties2positions[difficulty] = [(pos, way)]

    visited = np.zeros(shape, dtype="uint8")  # 0: nicht besucht, 1: als schnellstes Feld
                                              # besucht, 2: als normales (und schnellstes) Feld besucht
    visited[current_pos] = 1
    # Positionen von Feldern, die nur einmal besucht werden:
    one_visit_positions = [(next_pos, [])
                           for next_pos in current_pos.neighbor_fields(0, size-1, 0, size-1)
                           if field[next_pos] < 2 or difficulties[next_pos] == 255]
    # Positionen, bei denen Felder auf dem Weg u.U. doppelt besucht wurden
    double_visit_positions = []
    for remaining_charge in range(
                  initial_charge-1,
                  # siehe Lösungsidee für folgende Formel.
                  # remaining_charge darf nicht kleiner als 0 werden (max(…, -1)).
                  max(initial_charge+(-5 if initial_charge % 2 == 0 else -4), -1), -1):
        rand.shuffle(one_visit_positions)
        rand.shuffle(double_visit_positions)
        # Positionen zuerst betrachten, die sich bereits auf Wegen befinden, für die also
        # kein zusätzliches Feld als Weg markiert werden muss
        one_visit_positions.sort(key=lambda pos_and_way: 0 if field[pos_and_way[0]] == 1 else 1)
        double_visit_positions.sort(key=lambda pos_and_way: 0 if field[pos_and_way[0]] == 1 else 1)
        # Felder, die als nächstes (mit remaining_charge=remaining_charge-1) besucht werden
        next_one_visit_positions = []
        next_double_visit_positions = []
        # nur einmal besuchte Felder
        for pos, old_way_to_pos in one_visit_positions:
            if visited[pos] == 0:
                visited[pos] = 1
                if remaining_charge == 0 or (remaining_charge < initial_charge-2 and
                                             remaining_charge % 2 == 0):
                    # Auch mit Wegen mit nur einmal besuchten Feldern kann es vorkommen, dass hier
                    # bereits mögliche nächste Felder gefunden werden (z.B. wenn initial_charge < 3,
                    # dann kann remaining_charge == 0 auch hier schon zu True auswerten oder wenn
                    # initial_charge-4 == remaining_charge: Mit dieser verbleibenden Ladung gibt es
                    # sowohl nächste Felder mit Wegen, auf denen Felder nur einmal besucht werden,
                    # als auch Felder, bei denen alle Felder der Wege nur einmal besucht werden)
                    add_new_position(pos, old_way_to_pos, remaining_charge == 0)
                if difficulties[pos] != 255:
                    way_to_pos = old_way_to_pos + [pos]
                    for next_pos in pos.neighbor_fields(0, size-1, 0, size-1):
                        if field[next_pos] < 2 or difficulties[next_pos] == 255:
                            # Auf einem Feld mit difficulties[pos] == 255 liegt bereits
```

```
                            # eine Batterie (d.h. is_valid_field_for_way gibt False zurück),
                            # mit dieser Position dürfte allerdings trotzdem
                            # add_new_position aufgerufen werden (es handelt sich um ein
                            # mögliches nächstes Feld). Von einem solchen Feld aus dürfen
                            # allerdings keine Nachbarfelder besucht werden. Dies verhindert
                            # obige Bedingung 'if difficulties[pos] != 255:'.
                            if visited[next_pos] == 0:
                                next_one_visit_positions.append((next_pos, way_to_pos))
                            else:
                                # Feld kann doppelt besucht werden
                                next_double_visit_positions.append((next_pos, way_to_pos))
        # Felder, die doppelt besucht werden
        for pos, old_way_to_pos in double_visit_positions:
            assert visited[pos] != 0
            if visited[pos] == 1:
                if pos not in old_way_to_pos:  # Eine neue Batterie darf nicht auf ein Feld gelegt
                                               # werden, das als Weg zu diesem Feld genutzt wird.
                    # Das Feld wird nur als besucht markiert, wenn auch wirklich ein valider
                    # Weg zu diesem Feld gefunden wurde. Ansonsten gibt es vielleicht
                    # andere Wege zu diesem Feld, die valide sind, aber nicht mehr betrachtet
                    # würden, weil visited[pos] bereits auf 2 gesetzt ist.
                    visited[pos] = 2
                    if remaining_charge % 2 == 0 and remaining_charge < initial_charge-2:
                        # Das Feld ist ein mögliches nächstes Feld
                        add_new_position(pos, old_way_to_pos, remaining_charge == 0)
                if difficulties[pos] != 255:
                    way_to_pos = old_way_to_pos + [pos]
                    for next_pos in pos.neighbor_fields(0, size-1, 0, size-1):
                        if field[next_pos] < 2 or difficulties[next_pos] == 255:
                            next_double_visit_positions.append((next_pos, way_to_pos))
        one_visit_positions = next_one_visit_positions
        double_visit_positions = next_double_visit_positions
    # ab hier keine Felder doppelt besuchen
    for remaining_charge in range(remaining_charge-1,  # beginne da, wo die vorherige
                                  -1, -1):             # Schleife aufgehört hat
        rand.shuffle(one_visit_positions)
        one_visit_positions.sort(key=lambda pos_and_way: 0 if field[pos_and_way[0]] == 1 else 1)
        next_one_visit_positions = []
        for pos, old_way_to_pos in one_visit_positions:
            if not visited[pos]:
                visited[pos] = 1
                if remaining_charge % 2 == 0:
                    add_new_position(pos, old_way_to_pos, remaining_charge == 0)
                if difficulties[pos] != 255:
                    way_to_pos = old_way_to_pos + [pos]
                    for next_pos in pos.neighbor_fields(0, size-1, 0, size-1):
                        if field[next_pos] < 2 or difficulties[next_pos] == 255:
                            next_one_visit_positions.append((next_pos, way_to_pos))
        one_visit_positions = next_one_visit_positions
    return choose_next_battery(rand, field, normal_difficulties2positions,
                               fastest_difficulties2positions,
                               probability_do_multi_visit, probability_take_fastest)

def get_free_neighbor_fields(pos: Vector2):
    return (p for p in pos.neighbor_fields(0, size-1, 0, size-1) if field[p] < 2)

shape = (size, size)
field = np.zeros(shape, dtype="uint8")  # 0: freies Feld, 1: Weg oder Roboterstartfeld,
                                        # 2: Batteriefeld
difficulties = np.empty(shape, dtype=int)
difficulties.fill(1)
# Roboterstartfeld (in der Mitte des Feldes)
pos = Vector2(size//2, size//2)
charge = get_random_charge(rand)
print("robot battery charge is", charge)
batteries = [(pos, charge)]  # Alle Batterien (inkl. anfängliche Bordbatterie)
field[pos] = 1
solution = [pos]
# {<Position>: <Ladung, die der Roboter von dieser Position bei nochmaligem Besuchen aufnimmt>}:
multi_visit_fields = {}
```

```
    old_charge = charge
    old_pos = pos
    # Folgende Variable speichert den Index einer Batterie in batteries. Sie wird dazu genutzt, um
    # festzustellen, die Ladung welcher Batterie nachträglich reduziert werden muss, wenn eine Batterie
    # nicht mehrfach besucht werden konnte. Die Ladung wird dann so reduziert, dass sie gleich der Ladung
    # ist, mit der tatsächlich das nächste Feld gesucht wurde (old_charge-multi_visit_charge, s.u.).
    battery_num_from_which_charge_originates = 0
    while len(batteries) < battery_count:
        print("\nBATTERY")
        # Entscheide, ob diese Batterie mehrfach besucht werden soll
        if old_charge > 1 and rand.random() < probability_mark_as_multi_visit:
            print("the field for this battery should be multi-visited")
            multi_visit_charge = rand.randint(1, old_charge - 1)
            print("  reduced charge to search new battery positions from", old_charge, "to",
                  old_charge-multi_visit_charge)
            # Reduziere die Ladung um multi_visit_charge (mit old_charge wird das nächste Feld
            # gesucht)
            old_charge -= multi_visit_charge
            is_multi_visit = True
        else:
            is_multi_visit = False
        print("search ways from", old_pos, "with charge", old_charge)
        try:
            pos, way = get_next_battery_position(old_pos, old_charge)
        except NoNewPositionsError:
            print("no free way")
            break
        print("position for battery:", pos)
        solution.extend(way)
        solution.append(pos)
        # Markiere Felder als besetzt durch Weg
        for way_pos in way:
            field[way_pos] = 1

        old_battery_num_from_which_charge_originates = battery_num_from_which_charge_originates

        if difficulties[pos] == 255:
            # Dieses Feld wird nicht das erste Mal besucht. Auf das Feld wird also keine neue
            # zufällige Ladung gelegt,sondern es wird die Ladung aus multi_visit_fields ermittelt,
            # die auf diesem Feld übrig geblieben ist.
            assert field[pos] != 0
            print("position was visited before")
            charge, battery_num_from_which_charge_originates = multi_visit_fields[pos]
            print("  charge on this field is", charge)
            del multi_visit_fields[pos]
            difficulties[pos] = 0
        else:
            charge = get_random_charge(rand)
            # a battery with charge 'charge' is placed at pos
            print("  set charge on this field to", charge)
            assert field[pos] == 0
            field[pos] = 2
            battery_num_from_which_charge_originates = len(batteries)  # die Batterie, die ...
            batteries.append((pos, charge))  # ... hier angefügt wird

        if is_multi_visit:
            assert pos not in multi_visit_fields
            multi_visit_fields[pos] = (multi_visit_charge,
                                       old_battery_num_from_which_charge_originates)
            difficulties[pos] = 255
            print("this field should be multi-visited, charge is going to be",
                  multi_visit_charge, "next time")
        old_charge = charge
        old_pos = pos

    # SUCHE EINEN WEG, UM DIE LETZTE BATTERIE ZU ENTLADEN

    def search_neighbor_field_with_way(pos):
        # Sucht nach einem Nachbarfeld, das möglichst als Weg benutzt wird
        # return: (Nachbarfeld, ob das Nachbarfeld als Weg benutzt wird)
        neighbor_pos = None
```

```
        for neighbor in get_free_neighbor_fields(pos):
            if (f := field[neighbor]) == 1:
                return neighbor, True
            elif f == 0:
                neighbor_pos = neighbor
        return neighbor_pos, False

    if charge <= 2 or pos == batteries[0][0]:  # batteries[0][0] ist Position des Roboterstartfeldes
        # Es wird nur ein freies Nachbarfeld benötigt.
        # Dieses Nachbarfeld sollte wenn möglich schon als Weg genutzt werden, dann muss kein neues
        # Feld als Weg markiert werden. Dies ist sinnvoll, weil aus field (weiter unten) alle Zeilen
        # und Spalten am Rand entfernt werden, die nur 0en enthalten, die Feldgröße wird also auf ein
        # Minimum reduziert. Damit das Spielbrett nicht durch den Weg der Batterie größer wird, nutzt
        # die letzte Batterie wenn möglich bereits als Weg markierte Felder.
        neighbor = search_neighbor_field_with_way(pos)[0]
        if neighbor is not None:
            field[neighbor] = 1
        else:
            # Für die letzte Batterie konnte kein Weg gefunden werden, die Batterie muss entfernt
            # werden.
            del batteries[-1]
    else:
        # Eines der Nachbarfelder muss auch ein freies Nachbarfeld haben. Auch hier werden
        # zwei Felder ausgewählt, die möglichst bereits als Weg markiert wurden.
        neighbor_field1 = None
        neighbor_field2 = None
        found_one_field_with_way = False
        for neighbor in get_free_neighbor_fields(pos):
            if (f := field[neighbor]) == 1:
                neighbor_neighbor, is_way = search_neighbor_field_with_way(neighbor)
                neighbor_field1 = neighbor
                neighbor_field2 = neighbor_neighbor
                if is_way:
                    break
                found_one_field_with_way = True
            elif f == 0:
                if found_one_field_with_way:
                    continue
                neighbor_neighbor, is_way = search_neighbor_field_with_way(neighbor)
                neighbor_field1 = neighbor
                neighbor_field2 = neighbor_neighbor
                if is_way:
                    found_one_field_with_way = True
        if neighbor_field1 is not None and neighbor_field2 is not None:
            field[neighbor_field1] = 1
            field[neighbor_field2] = 1
        else:
            del batteries[-1]

    # Reduziere die Ladung von allen Batterien, deren Ladung für die Suche nach einem Feld für die
    # nächste Batterie reduziert wurden, weil die nächste Batterie nochmals besucht werden sollte,
    # aber die nächste Batterie nicht besucht werden konnte.
    for pos, (multi_visit_charge, battery_num_from_which_charge_originates) in
                                                                        multi_visit_fields.items():
        pos, old_charge = batteries[battery_num_from_which_charge_originates]
        new_charge = old_charge - multi_visit_charge
        batteries[battery_num_from_which_charge_originates] = (pos, new_charge)

    # Ermittle, wie viele Spalten und Zeilen vom Spielbrett leer sind (d.h. dort befindet sich keine
    # Batterie und die Felder werden nicht als Wege genutzt) und entfernt werden können.
    for start_x in range(size):
        if not np.all(field[:, start_x] == 0):
            break
    for start_y in range(size):
        if not np.all(field[start_y, :] == 0):
            break
    for end_x in range(size-1, -1, -1):
        if not np.all(field[:, end_x] == 0):
            break
    for end_y in range(size-1, -1, -1):
        if not np.all(field[end_y, :] == 0):
```

```
            break

    # [...] Speichere die Spielsituation.
```

Die Funktion `choose_next_battery` wählt eine neue Batterie aus. Befindet sich `255` in einem der übergebenen `dicts`, dann wird mit der (ebenfalls übergebenen) Wahrscheinlichkeit `probability_do_multi_visit` eine Position aus denen mit Schwierigkeitsgrad `255` ausgewählt. Ansonsten wird ein Schwierigkeitsgrad mit einem Aufruf von `choose_difficulty` ausgewählt. Für jede Position mit dem gewählten Schwierigkeitsgrad wird bestimmt, wie viele Felder neu als Weg markiert werden müssten (`field[pos] == 0`). Die Position mit den wenigsten neu als Weg markierten Feldern wird zurückgegeben.

5 Literaturverzeichnis

[1] C. Y. Lee, “An Algorithm for Path Connections and Its Applications,” *IEEE Trans. Electron. Comput.*, vol. EC-10, no. 3, pp. 346-365, Sep. 1961, doi: 10.1109/TEC.1961.5219222

[2] R. Bellman, “Dynamic Programming Treatment of the Travelling Salesman Problem,” *J. ACM 9*, vol. 9, no. 1, pp. 61-63, Jan. 1962, doi: 10.1145/321105.321111