Aufgabe 1: Stromrallye

Teilnahme-Id: 55174

Bearbeiter/-in dieser Aufgabe:

Nils Weißer

Inhaltsverzeichnis

1 Lösungsidee....3
1.1 Theoretische Beschreibung der Aufgabenstellung....3
1.2 Grundregeln....3
1.2.1 Vor dem Spiel....3
1.2.2 Während des Spiels....4
1.2.3 Weitere Regeln....4
1.3 Der Algorithmus....5
1.3.1 A*-Algorithmus....5
1.3.2 Finden der Reihenfolge der Ersatzbatterien....6
1.4 Laufzeitanalyse....6
1.5 Stromrallye: Das Spiel....7
2 Umsetzung....8
2.1 Umsetzung Teilaufgabe a)....8
2.2 Umsetzung Teilaufgabe b)....12
3 Erweiterungsausblicke....13
3.1 Hindernisse....13
3.2 Portale....13
4 Beispiele....14
4.1 Einführung....14
4.2 Beispiel 1....14
4.3 Beispiel 2....16
4.4 Beispiel 3....16
4.5 Beispiel 4....16
4.6 Beispiel 5: Schwierigkeitsgrad Einfach....16
4.7 Beispiel 6: Schwierigkeitsgrad Mittel....17
4.8 Beispiel 7: Schwierigkeitsgrad Schwer....17
4.9 Beispiel 8: Sonderfall 1....18
4.10 Beispiel 9: Sonderfall 2....18
5 Quelltext....19
5.1 Variablenliste Teilaufgabe a)....19
5.2 Programmcode Teilaufgabe a)....20
5.3 Programmcode Teilaufgabe b)....24

1 Lösungsidee

1.1 Theoretische Beschreibung der Aufgabenstellung

Das Problem dieser Aufgabe besteht darin, dass ein Weg gefunden werden soll, der alle Ersatzbatterien in einer bestimmten Reihenfolge erreicht, sodass alle Ladungen am Ende verbraucht sind. Um die einzelnen Teilstrecken, also die Wege zwischen den Batterien, zu ermitteln, wird ein Wegfindungsalgorithmus benötigt. Dann müssen die Teilstrecken noch so verknüpft werden, dass die Ersatzbatterien in der richtigen Reihenfolge abgelaufen werden.

1.2 Grundregeln

Hier werden zunächst einige Grundregeln und Abbruchbedingungen für eine zielgerichtete Lösungen des Spiel erläutert. Diese Regeln sind unterteilt in: „Vor dem Spiel“, „Während des Spiels“ und „Weitere Regeln“. Es sind vermutlich noch deutlich mehr Regeln zu finden, um das Programm weiter zu optimieren. Mit höherer Anzahl von Regeln & Abbruchbedingungen kann das Spiel in Einzelfällen schneller gelöst werden, jedoch bedeutet es für den Großteil der Fälle eine höhere Laufzeit.

1.2.1 Vor dem Spiel

Dies sind Abbruchbedingungen, die bereits vor dem Spiel geprüft werden, also nach Einlesen der Daten.

1. Wenn eine Ersatzbatterie eine Ladung von 1 hat, muss sie mindestens eine andere Ersatzbatterie auf einem benachbarten Feld haben, sonst kann es nur die letzte Batterie sein.
2. Wenn es mindestens zwei Ersatzbatterien mit einer Ladung von 1 und ohne Nachbar gibt, ist die eingelesene Spielsituation nicht lösbar.
3. Wenn eine Ersatzbatterie von sich aus keine andere Ersatzbatterie erreichen kann, muss es entweder die letzte Batterie sein oder der Roboter muss direkt wieder auf das gleiche Feld zurück. In diesem Fall müsste mit der zuvor auf dem Punkt abgelegten Batterie die nächste Batterie erreicht werden. Sonst wäre die Spielsituation nicht lösbar.
4. Wenn eine Ersatzbatterie nicht von einer anderen Ersatzbatterie oder dem Startpunkt des Roboters erreicht werden kann, ist die Spielsituation nicht lösbar.

1.2.2 Während des Spiels

Dies sind Abbruchbedingungen, die erst im Laufe des Spiels erkannt werden können.

1. Wenn der Roboter mit der momentanen Ladung keine Ersatzbatterie erreichen kann, ist es nicht von dieser Situation aus lösbar (vorausgesetzt es gibt noch Ersatzbatterien, die erreicht werden müssen).
2. Wenn die Ladung des Roboters 0 ist, ist das Spiel nicht über diesen Weg lösbar (außer die Ladung erreicht gerade 0 wenn der Roboter ein Feld mit einer Ersatzbatterie erreicht).
3. Wenn der Roboter auf ein Feld geht, auf dem eine Ersatzbatterie mit der Ladung 0 liegt, ist das Spiel zu Ende.
4. Wenn der Mindestabstand vom aktuellen Standpunkt bis zu der zu erreichenden Batterie größer ist als die momentane Ladung, ist die Ersatzbatterie von diesem Punkt aus nicht zu erreichen.
5. Wenn der Mindestabstand vom aktuellen Standpunkt zu einer Ersatzbatterie größer ist als die Ladung aller Ersatzbatterien, ausgeschlossen der zu erreichenden Ersatzbatterie, zusammen mit der aktuellen Ladung des Roboters, kann das Spiel nicht mehr gelöst werden.
6. Wenn eine Ersatzbatterie umgeben von anderen Ersatzbatterien oder der Wand ist, also alle vier Plätze direkt neben ihr belegt sind, kann die Batterie nicht erreicht werden, außer der Roboter steht momentan auf einem dieser Nachbarpunkte.

1.2.3 Weitere Regeln

Hier sind weitere Regeln zu finden, die eher das Spiel betreffen und sich nicht auf die Abbruchbedingungen konzentrieren.

1. Ein Umweg verbraucht immer eine Ladung von einem Vielfachen von 2.
2. Es müssen mindestens so viele Wege zwischen den Batterien gegangen werden, wie es Ersatzbatterien gibt.
3. Bei ungerader Ladung kann der Roboter nicht direkt wieder auf das gleiche Feld gehen ohne das eine Restladung übrig bleibt.
4. Wenn der Mindestabstand zwischen einer Batterie und dem momentanen Standpunkt gerade ist, muss die momentane Ladung auch gerade sein um den Punkt mit einem Rest von 0 zu erreichen. Das selbe gilt für ungerade Mindestabstände und ungerade Ladungen.

1.3 Der Algorithmus

Wie in 1.1 beschrieben, wird zunächst ein Wegfindungsalgorithmus gebraucht um die einzelnen Strecken zwischen den Batterien zu ermitteln. Da es bei der Wegfindung zwischen zwei Punkten

bei dieser Problemstellung zu Hindernissen, in Form von anderen Ersatzbatterien, kommen kann, kann nicht einfach der kürzeste Weg genommen werden. Um die Hindernisse mit einzubeziehen, wird der A*-Algorithmus benutzt.

1.3.1 A*-Algorithmus

Der A*-Algorithmus wird verwendet um den kürzesten Pfad zwischen zwei Punkten zu berechnen. Er nutzt Heuristiken, um gezielt nach dem kürzesten Weg zu suchen. Der Algorithmus findet, sofern sie existiert, immer die perfekte Lösung.
Dieser Algorithmus basiert auf dem Dijkstra-Algorithmus. Bei dem Dijkstra-Algorithmus werden, ausgehend von dem Startknoten, alle mit ihm verbundenen Knoten verarbeitet. Dann von diesen Knoten wieder die folgenden Knoten. Die Kosten für den Weg werden immer summiert, sodass der Weg mit den wenigsten Kosten priorisiert wird und als nächstes verfolgt wird. Da dieser Algorithmus jedoch bei einem Netz, bei dem jede Kante die gleichen Kosten hat, keinen Sinn ergibt, muss eine erweiterte Vorgehensweise benutzt werden.
Um zielgerichtet nach dem optimalen Weg zu suchen, wird auf jeden Wegknoten eine Formel angewandt, die ungefähr bestimmt, wie weit der Knoten von dem Zielpunkt entfernt ist. Dieser berechnete Wert wird als f-Wert bezeichnet. Er setzt sich aus dem g-Wert, die Distanz des Knotens zum Startknoten, und dem h-Wert, die geschätzte Distanz zum Zielknoten zusammen. Der h-Wert wird mit dem Satz des Pythagoras berechnet. Wenn die Koordinaten eines Knotens (x_1,y_1) und die Koordinaten des Zielknotens (x_2,y_2) sind, wird der Abstand mit der Formel $\sqrt{((x_1-x_2)^2+(y_1-y_2)^2)}$ berechnet. Da die Kosten zwischen den Knoten bei dieser Aufgabenstellung immer 1 sind, ist der g-Wert einfach die Anzahl der bereits gegangenen Knoten. Im Vergleich zu dem Dijkstra-Algorithmus, in dem nur mit dem g-Wert gearbeitet wird, wird bei dem A*-Algorithmus auch noch die Schätzfunktion h betrachtet.
Jeder Knoten des Spielfelds befindet sich bei dem A*-Algorithmus in einer von drei Bereichen.
1. Der Knoten ist unbekannt
2. Der Knoten ist in der Warteschlange
3. Der Knoten ist bereits abgearbeitet

Die Warteschlange wird auch „open list“ genannt. Der Algorithmus wird begonnen, indem der Startknoten zur „open list“ hinzugefügt wird. Ganz oben in der Warteschlange, ist immer der Knoten mit dem niedrigsten f-Wert. Er wird also als nächstes betrachtet. Wenn ein Knoten betrachtet wurde, wird er in die „closed list“ verschoben. Seine Nachbarn werden, sofern sie noch unbekannt sind, in die Warteschlange mit ihrem berechneten f-Wert verschoben. Wenn der Nachbar sich bereits in der Warteschlange befindet, wird überprüft, ob der Nachbar nun einen geringeren f-Wert hat, ist

dies der Fall ersetzt er den bereits in der Warteschlange gespeicherten Knoten, wenn nicht wird dieser Nachbar übersprungen. Der Algorithmus hat den Weg gefunden, wenn der Knoten der ganz vorne in der Warteschlange steht, der Zielknoten ist.

1.3.2 Finden der Reihenfolge der Ersatzbatterien

Zum finden der richtigen Reihenfolge, in der der Roboter die Ersatzbatterien abläuft wird ein einfacher Bruteforce-Algorithmus verwendet. Vom Startpunkt aus werden alle Ersatzbatterien, die im Bereich der Ladung sind, mit dem kürzesten Weg angelaufen. Überschüssige Ladungen werden zunächst vernachlässigt. Sie werden einfach abgelegt. Von den erreichten Ersatzbatterien werden wieder alle Batterien, die im Bereich der Ladung sind angelaufen. Dies geschieht, bis eine der unter 1.2 genannten Abbruchbedingungen eintritt oder jede Ersatzbatterie mindestens einmal erreicht wurde. Wenn dies der Fall ist, wird zunächst überprüft, ob alle übrig gebliebenen Ladungen ein Vielfaches von 2 sind, oder 0 sind. Wenn nicht, wird weiter nach einem Weg gesucht. Wenn alle übrig gebliebenen Ladungen ein Vielfaches von 2 sind, oder 0 sind, wird zunächst überprüft, ob die restliche Ladung, die der Roboter noch hat, jedoch nur noch verbrauchen muss, da keine Batterie mehr zu erreichen ist, verbraucht werden kann. Dafür wird in alle Richtungen gegangen, bis die Ladung leer ist. Wenn die Ladung nicht verbraucht werden kann, wird wieder nach einem anderen Weg gesucht. Wenn schon, wird überprüft, ob die übrig gebliebenen Ladungen noch verbraucht werden können. Dafür wird die letzte Teilstrecke bis zu dem Punkt mit der übrig gebliebenen Ladung betrachtet. Wenn die Teilstrecke mindestens eine Länge von 3 hat, kann die Ladung durch einfaches vor- und zurückgehen verbraucht werden. Wenn dies nicht möglich ist, muss nach einem neuen Weg gesucht werden. Wenn die Ladungen verbraucht werden konnten, wurde der fertige Weg gefunden.

1.4 Laufzeitanalyse

Wenn das eingelesene Beispiel von einer der in 1.2.1 vorgestellten Bedingungen schon vor dem eigentlichen Programm abgefangen wird, ist die Laufzeit natürlich minimal. Das wäre das Best-Case Szenario. Der wichtigste Faktor für die Laufzeit bei diesem Algorithmus ist die Anzahl der Ersatzbatterien. Bei einer sehr hohen Anzahl von Ersatzbatterien, werden für jeden Punkt immer eine sehr hohe Anzahl an weiterführenden Wegen berechnet, was dazu führt, dass die Laufzeit sehr hoch ist. Außerdem spielt die Höhe der Ladungen der Ersatzbatterien eine Rolle, da von einem Punkt nur die Ersatzbatterien verfolgt werden, die in der Reichweite des Roboters liegen. Wenn also die Ladungen besonders hoch sind, gibt es mehr mögliche Wege. Die Worst-Case Laufzeit für

eine Teilstreckensuche mit dem A*-Algorithmus beträgt $O(|V|^2)$. V steht für die Menge der Knoten auf dem Spielfeld.

1.5 Stromrallye: Das Spiel

In Teilaufgabe b) soll ein Algorithmus erstellt werden, der eine Spielsituation erstellt, die lösbar ist, aber schwer für menschliche Spieler. Für diese Aufgabe muss zunächst definiert werden, welche Eigenschaften einer Spielsituation schwer oder leicht machen. Dazu zählen:

1. Die Größe des Spielfelds
2. Der Anteil der Belegung von Ersatzbatterien auf dem Spielfeld
3. Die Weglänge zwischen den Ersatzbatterien

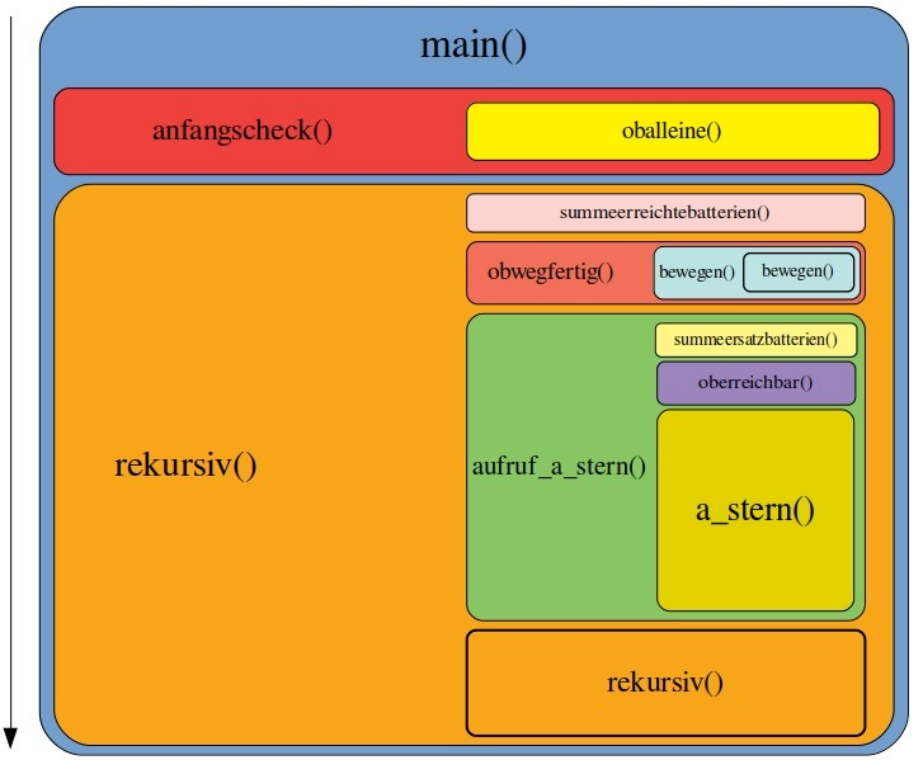
Drei Schwierigkeitsgrade sind für das Spiel entworfen worden:

1. Einfach: Größe = 5, Anteil der Belegung: zwischen 8% und 20% (entspricht 2 und 5 Batterien)
2. Mittel: Größe = 10, Anteil der Belegung: zwischen 10% und 15% (entspricht 10 und 15 Batterien)
3. Schwer: Größe = 20, Anteil der Belegung: zwischen 10% und 15% (entspricht 40 und 60 Batterien)

Der Algorithmus basiert auf dem Algorithmus der ersten Teilaufgabe.
Zunächst wird ein Startpunkt zufällig bestimmt. Da bei diesem Algorithmus der Weg von hinten durchlaufen wird, ist dieser Startpunkt beim richtigen Spiel der letzte Punkt. Dann werden zufällig Ersatzbatterien auf dem Spielfeld verteilt. Jetzt wird vom Startpunkt aus eine Ersatzbatterie zufällig ausgewählt, solange sie erreichbar ist. Dann wird der Weg zu ihr mit dem bereits in Teilaufgabe a) vorgestellten A*-Algorithmus berechnet. Von dieser Ersatzbatterie wird wieder eine andere Ersatzbatterie ausgewählt. Dies wird immer weiter fortgeführt, bis alle Ersatzbatterien mindestens einmal erreicht wurden. Sie können aber auch mehrfach erreicht werden. Um die Ladung der Ersatzbatterien zu bestimmen, wird immer der Abstand zwischen dem aktuellen Startpunkt und der nächsten Ersatzbatterie ermittelt. Dieser Abstand ist nun die Ladung für die zu erreichende Ersatzbatterie. Wenn die Ersatzbatterie bereits eine Ladung hat, da sie bereits erreicht wurde, wird die Ladung durch den neuen Abstand ersetzt. Die alte Ladung wird dann mit auf die Ladung der nächsten Ersatzbatterie addiert.

2 Umsetzung

2.1 Umsetzung Teilaufgabe a)



Diese Abbildung zeigt eine Übersicht der Python3-Implementierung des Algorithmus. In dieser Übersicht werden alle Funktion als Rechtecke mit verschiedenen Farben dargestellt. Der zeitliche Ablauf des Programms entspricht dem Pfeil an der linken Seite der Abbildung. Wenn eine Funktionen innerhalb einer anderen Funktion geschrieben ist, wird die innere Funktion von der äußeren aufgerufen. Außerdem steht die Größe der Rechtecke in Zusammenhang mit der Relevanz der Funktionen. Wenn ein Rechteck besonders groß ist, ist die Funktion wichtiger als wenn ein Rechteck eher klein ist.

Zunächst liest das Programm die Datei ein und initialisiert einige Variablen mit den Daten der Datei. Dann wird jeder in jede Liste mit den Daten der Ersatzbatterien in der Liste `koordinaten_eingelesen[]` ein weiteres Attribut eingefügt. Dieser Wert wird auf 0 gesetzt und wird später benutzt um zu überprüfen ob diese Ersatzbatterie bereits erreicht wurde.

Dann wird die Funktion main() aufgerufen. In ihr wird die Funktion anfangscheck() aufgerufen.

Die Funktion anfangscheck() beinhaltet einige Tests bevor das Hauptprogramm ausgeführt wird. Diese Tests schließen direkt einige eingelesene Spielsituationen aus. Zuerst wird überprüft, ob und wenn ja wie viele, Ersatzbatterien es mit einer Ladung von 1 gibt, die keine Nachbarn haben.

Wenn es mehr als eine Ersatzbatterie mit einer Ladung von 1 gibt, wird das Programm abgebrochen, da diese Spielsituation nicht mehr lösbar ist. Dann wird die Funktion oballeine() aufgerufen, die auch eine Überprüfung darstellt.

In der Funktion oballeine() werden mit Hilfe einer verschachtelten for-Schleife alle Ersatzbatterien miteinander verglichen und ihr minimaler Abstand. ohne Betrachtung der anderen Ersatzbatterien, voneinander ermittelt. Wenn eine Ersatzbatterie von keiner anderen Batterie oder dem Startpunkt erreicht werden kann, wird das Programm abgebrochen, da kein Weg mehr gefunden werden kann, der diese Batterie beinhaltet.

Nach dem diese Anfangsüberprüfungen durchgeführt wurden, geht es weiter mit dem richtigen Programm. Die Funktion rekursiv() wird in main() aufgerufen.

Die Funktion rekursiv() wird, wie der Name schon sagt, rekursiv aufgerufen. Das führt dazu, dass die ersten Schritte dieser Funktion für den ersten Aufruf wenig Sinn machen und erst ab dem zweiten Aufruf relevant werden, da die Daten des vorherigen Aufrufs übergeben werden. In dieser Funktion wird als Erstes die lokale Liste `lokliste[]` um den vorherigen Weg, also den Weg von der vorherigen Ersatzbatterie bis zum momentanen Standpunkt, erweitert. Dann wird mit einer for-Schleife über die Liste `koordinaten[]` iteriert und nach dem aktuellen Standpunkt gesucht. Wenn er gefunden wird, wird die Ladung in der Liste durch die restliche Batterie, also die Batterie, die abgelegt werden soll, ersetzt. Außerdem wird der Standpunkt als „bereits erreicht“ markiert, in dem der Wert an der 4. Stelle der Liste auf 1 gesetzt wird. Nun folgt die Abbruchbedingung. Dafür wird zunächst die Funktion summeerreichtebatterien() aufgerufen.

In der Funktion summeerreichtebatterien() wird der Integer `summe` mit der globalen Variable `anzahl_batterien` initialisiert. Dann wird über die Liste `koordinaten` iteriert, und somit der Wert an der 4. Stelle jeder Ersatzbatterie von der Summe abgezogen. Dieser Wert enthält, ob die Ersatzbatterie bereits im Laufe des Weges erreicht wurde oder nicht. Wenn `summe` am Ende der for-Schleife einen Wert von 0 hat, bedeutet es, dass jede Ersatzbatterie mindestens einmal erreicht wurde. Dann wird mit Hilfe der modulo-Operation überprüft, ob die übrig gebliebenen Ladung alle durch 2 teilbar sind. Ist dies nicht der Fall wird 0 an die Funktion rekursiv() übergeben und die Abbruchbedingung somit übersprungen und weiter nach einem Weg gesucht. Wenn die genannten Überprüfungen jedoch stimmen, wird 1 zurückgegeben und in der Funktion rekursiv() wird die Funktion obwegfertig() aufgerufen.

Die Funktion obwegfertig() verfolgt zunächst das Ziel, einen Weg für die letzte Ladung zu finden, also für die Ladung, die der Roboter erhält, wenn er die letzte Ersatzbatterie erreicht und kein bestimmtes Ziel mehr hat, außer die Ladung zu verbrauchen. Dafür wird die Liste `richtungen[]`

angelegt, in die vier möglichen x- und y-Änderungen, sodass der Roboter sich um ein Feld bewegt, gespeichert werden. Außerdem wird die globale Liste `letztekoordinaten[]` deklariert. Jetzt wird die Funktion bewegen() mit den einzelnen Richtungen als Übergabeparameter aufgerufen.

In der Funktion bewegen() ist zunächst eine Abbruchbedingung definiert. Wenn der boolean `stop` auf True gesetzt ist, wird direkt ein return ausgeführt. Dann wird überprüft ob die Ladung des momentanen Standpunkts den Wert 0 hat. Ist dies der Fall, wird die Variable `letztekoordinaten[]` auf den Wert der lokalen Liste `bewegen_liste[]`.Diese Liste enthält die Koordinaten von dem Weg zwischen der letzten Ersatzbatterie, bis zu dem Punkt an dem die letzte Batterieladung aufgebraucht ist. Außerdem wird der boolean `stop` auf True gesetzt. Wenn die Ladung des momentanen Standpunktes jedoch nicht 0 ist, geht es weiter. Es wird die übergebene Richtung zu den Koordinaten des momentanen Standpunktes addiert und die Ladung um 1 verringert. Außerdem werden die neuen Koordinaten zu der lokalen Liste `bewegen_liste[]` hinzugefügt, um am Ende den fertigen Weg gespeichert zu haben. Dann wird die Funktion `bewegen()` rekursiv wieder mit den vier Richtungen aufgerufen. Außerdem wird der neue Standpunkt übergeben. Wenn du Funktion zu Ende ist, geht es wieder zurück in die Funktion obwegfertig(). Wenn kein Weg gefunden wurde, wird die Funktion beendet und es wird nach einem anderen Weg gesucht. Wenn ein Weg gefunden wurde, wird eine weitere Überprüfung durchgeführt. Dadurch, dass bei der Wegsuche Ersatzbatterien, die eine Ladung von einem Vielfachen von 2 haben, liegen geblieben sein können, muss nun geschaut werden, ob man diese Batterien noch aufbrauchen kann. Zunächst wird in der lokalen Liste `lokliste[]`, die übergeben wurde, nach Koordinaten gesucht, auf denen noch eine Ersatzbatterie liegt, die eine Ladung von einem Vielfachen von 2 hat. Es wird nach dem letzten Eintrag dieses Koordinatenpaares in der Liste `lokliste[]` gesucht. Wenn es gefunden wurde, wird überprüft, ob auf mindestens einem der beiden Punkte davor im Weg eine Ersatzbatterie lag. Ist dies der Fall, kann die übrige Ladung nicht einfach verbraucht werden und es muss nach einem neuen Weg gesucht werden. Wenn jedoch keine der beiden vorherigen Punkte im Weg eine Ersatzbatterie beherbergen, werden diese beiden Punkte einfach so oft an dieser Stelle kopiert, wie die Ladung geteilt durch zwei. Das führt dazu, dass der Roboter immer vor und zurück geht und somit die Ladung dann nach diesem Ablauf aufgebraucht hat. War dies erfolgreich, wird in der Funktion rekursiv() nur noch die Ausgabe getätigt und dann das Programm beendet. Wenn noch nicht der Endweg gefunden wurde, geht es in der Funktion rekursiv() weiter.

Hier wird zunächst überprüft, ob der momentane Standpunkt bereits im Laufe des Programms erreicht wurde und somit schon alle möglichen nächsten Wege berechnet wurden. Wenn dies der Fall ist, werden die Wege einfach aus dem Dictionary `wege_dict{}` kopiert. Wenn der Punkt jedoch noch nicht erreicht wurde, wird die Funktion aufruf_a_stern() aufgerufen.

In der Funktion aufruf_a_stern() werden zunächst, die Listen `alle_im_feld[]`, in der alle möglichen Koordinaten gespeichert werden, `hindernisse[]`, in der die Koordinaten aller Ersatzbatterien gespeichert werden und `ersatzbatterien[]`, in die Koordinaten aller Ersatzbatterien gespeichert werden, die keine Ladung von 0 haben und nicht der Startpunkt sind. Dann wird mit einer for-Schleife über die Liste `ersatzbatterien[]` iteriert. Es wird für den aktuellen Endpunkt aus der Liste der minimale Abstand zum momentanen Standpunkt berechnet. Wenn die Summe aller Ersatzbatterien, die mit Hilfe der Funktion `summeersatzbatterien[]` berechnet wurde, mit der momentanen Ladung zusammen, unter dem Abstand liegt, wird ein return ausgeführt. Da eine Batterie nicht mehr erreichbar ist, muss nicht weiter gesucht werden. Wenn nur die momentane Ladung unter dem Abstand liegt, wird mit dem nächsten Endpunkt aus der Liste `ersatzbatterien[]` fortgeführt. Dann wird der momentane Standpunkt, sowie der Endpunkt der Funktion oberreichbar() übergeben.

In der Funktion oberreichbar() wird überprüft, ob der übergebene Endpunkt erreichbar ist, das heißt er nicht von Ersatzbatterien bzw. der Wand umschlossen ist. Ist dies der Fall, wird 1 zuückgegeben und somit der nächste Endpunkt in der Funktion aufruf_a_stern() behandelt. Wenn der Punkt erreichbar ist, wird er der Funktion a_stern() übergeben, die versucht durch die Implementierung des A*-Algorithmus einen Weg zwischen Start- und Endpunkt zu finden.

Die Funktion a_stern() beginnt damit, das sie die Listen `open_list[]` und `closed_list[]` erstellt und der momentanen Standpunkt zur `open_list[]` hinzufügt. Dann wird eine while-Schleife gestartet, die solange die Länge der Liste `open_list[]` größer als 0 ist, läuft. Dann wird die Variable `current` bestimmt, in der der Punkt mit dem momentan niedrigsten f-Wert aus der `open_list[]` gespeichert ist. Dieser Wert wird dann von der `open_list[]` entfernt und zur `closed_list[]` hinzugefügt. Wenn in der Variable `current` der gesuchte Endpunkt enthalten ist, wird der Weg zurückverfolgt, in dem jeder Punkt zu seinem „Parent“, also seinem vorherigen Punkt, zurückgeht und er in der Liste `weg[]` gespeichert wird. Dieser Weg wird dann zurückgegeben. Wenn `current` nicht die Koordinaten des Endpunktes enthält, geht es weiter mit der Bestimmung der Nachbarn von dem aktuellen Punkt. Wenn ein Nachbar ein Hindernis ist, bereits in der `closed_list[]` ist, oder außerhalb vom Feld ist, wird er übersprungen. Für alle diese Nachbarn werden dann die g-, h-, und f-Werte bestimmt. Wenn der Nachbar bereits in der `open_list[]` ist und dort einen niedrigeren g-Wert hat, wird auch dieser Nachbar übersprungen, wenn der g-Wert höher ist, wird der Wert in der Liste ersetzt. Zuletzt werden die Nachbarn zur `open_list[]` hinzugefügt. Wenn alle Wege von dem momentanen Standpunkt aus gefunden wurden, ist das Programm zurück in der Funktion rekursiv().

Dort wird erst einmal die Liste `alle_wege[]` in dem Dictionary `wege_dict{}` gespeichert. Hier kann wahlweise die Funktion matplot() benutzt werden, indem man sie nicht mehr auskommentiert. Mit dieser Funktion kann man die Wegsuche grafisch veranschaulicht beobachten (Achtung: Nur möglich wenn das Modul „matplotlib“ installiert ist). Dann wird über die Liste `alle_wege[]` iteriert. Zunächst wird der minimale Batterieverbrauch für den aktuellen Weg berechnet. Wenn dieser Batterieverbrauch größer ist, als die momentane Ladung, wird der Weg übersprungen und mit dem nächsten Weg fortgesetzt. Dann wird der neue Startpunkt bestimmt. Der neue Startpunkt ist der letzte Punkt des aktuellen Weges. Wenn der neue Startpunkt eine Ladung von 0 hat, wird mit dem nächsten Weg fortgesetzt. Wenn nicht, wird die Funktion rekursiv() erneut aufgerufen, jedoch mit den neuen Werten. Hierzu wird auch das Python BuiltIn-Modul „copy“ verwendet. Mit der Funktion deepcopy() aus diesem Modul kann man Listen kopieren, die weitere Listen beinhalten. Ohne diese Funktion würde man die inneren Listen nicht kopieren, sondern auf den gleichen Speicherplatz schreiben.

2.2 Umsetzung Teilaufgabe b)

Zunächst wird in der Funktion eingabe() mit einem User-Input der Schwierigkeitsgrad bestimmt. Dann wird die Funktion main() aufgerufen. Um die Zufallsziffern in einem bestimmten Bereich zu erhalten, wird die Funktion randint() aus dem Python Built-In Modul random verwendet. Damit werden dann die zunächst die Anzahl der Batterie und der Startpunkt bestimmt. Dann werden die Ersatzbatterien zufällig auf dem Feld verteilt, solange dort noch keine Ersatzbatterie liegt oder der Startpunkt auf diesem Punkt definiert ist. Danach wird die Funktion wegfinden() aufgerufen.

In dieser Funktion wird in einer while-Schleife zunächst die nächste Batterie mit der Methode choice() zufällig aus der Liste `ersatzbatterien[]` bestimmt. Wenn die ausgewählte Ersatzbatterie nicht der aktuellen Ersatzbatterie, auf der der Roboter momentan steht, entspricht, wird fortgeführt. Es werden die übrigen Ersatzbatterien in der Liste `hindernisse[]` abgespeichert, sodass sie nicht betreten werden. Dann wird der Index, der die Ersatzbatterien markiert, ob sie bereits erreicht wurden oder nicht, auf „1“ gesetzt. Von der aktuellen Ersatzbatterie zu der gerade neu bestimmten nächsten Ersatzbatterie wird nun der bereits in 2.1 erläuterte A*-Algorithmus verwendet, um den minimalen Abstand zu berechnen. Wenn kein Weg gefunden wurde, wird eine andere nächste Ersatzbatterie bestimmt. Sonst wird der Weg in die Liste `alle_wege[]` gespeichert. Dann gibt die Funktion mit einem return-Statement die nächste Batterie zurück. Sie ist jetzt für den erneuten Aufruf dieser Funktion der aktuellen Standpunkt. Nach jedem neuen Weg, der gefunden wurde, wird mit der Funktion summeerreichtebatterien() überprüft ob bereits alle Ersatzbatterien mindestens einmal erreicht wurden. Ist dies der Fall werden die Koordinaten sowie Ladungen der Ersatzbatterien mit der Funktion ladungen() bestimmt.

In der Funktion ladungen() wird über die Liste `alle_wege[]` iteriert. Wenn der Zielpunkt des aktuell behandelten Weges noch nicht in der Liste `koordinaten[]`, also der Liste der Ersatzbatterien, ist, werden die Koordinaten dieser Ersatzbatterie, sowie der Abstand des aktuellen Weges als Ladung zu dieser Liste hinzugefügt. Wenn die Ersatzbatterie jedoch bereits in der Liste ist, wird die Ladung an dieser Stelle durch die Länge des aktuellen Weges ersetzt. Der Wert der zuvor an der Stelle war, wird in der Variable `neue_ladung` gespeichert. Bei dem nächsten Weg wird nun die Ladung, die in `neue_ladung` gespeichert ist zu der eigentlich Ladung addiert. Somit hat man am Ende die fertige Liste `koordinaten[]` mit allen Ersatzbatterien, ihren Koordinaten und ihrer Ladung. Zuletzt wird noch die Funktion main_grafik() aufgerufen, die die erstellte Spielsituation nun anzeigt.

3 Erweiterungsausblicke

3.1 Hindernisse

Eine interessante Erweiterung wären Hindernisse, die auf dem Feld verteilt werden. Der Roboter müsste somit die Wege so finden, dass er alle Hindernisse umfährt. Dies führt zu einem erhöhten Schwierigkeitsgrad. Man könnte außerdem noch Hindernisse einbauen, die zwar überquert werden können, aber nur in eine Richtung. Beispielsweise könnte ein Hindernis nur in x-Richtung überquert werden.

3.2 Portale

Eine weitere Idee für eine Erweiterung der Aufgabenstellung, wären Portale. Auf dem Feld könnten Portale verstreut sein, die den Roboter bei Betreten eines Feldes mit einem Portal zu einem anderen Portal teleportieren würden.

4 Beispiele

4.1 Einführung

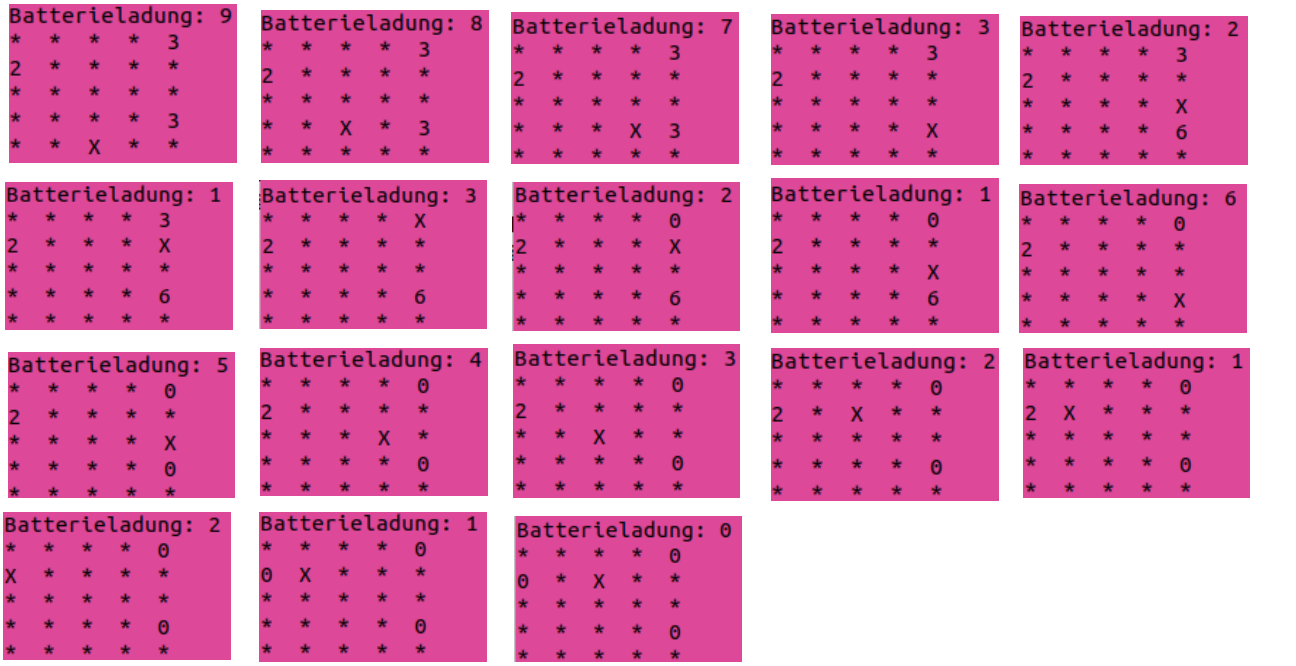
Aufgrund der sehr hohen Anzahl von Ersatzbatterien in den Beispielen „stromrallye1.txt“ und stromrallye2.txt“, konnten diese nicht von dem Algorithmus gelöst werden. Das Problem bei diesen Beispielen war, dass die letzte Batterie des Weges auf einem Nachbarpunkt des Startpunktes liegen muss. Dadurch war die Laufzeit zu hoch. Ein weiteres Problem dabei war, dass einzelne Batteriegruppen von dem eigentlichen Weg durch Ersatzbatterien mit der Ladung 0 abgeschottet wur-

den. Dort könnte man noch ein Algorithmus entwickeln, der erkennt ob Batterien nicht mehr erreicht werden können, da sie von Ersatzbatterien mit einer Ladung von 0 umgeben sind.
Deswegen folgen hier die Beispiele „stromrallye0.txt“, „stromrallye3.txt“, „stromrallye4.txt“, „stromrallye5.txt“, sowie einige Beispiele, die durch das Programm aus Teilaufgabe b) erstellt wurden.
Die angegeben Laufzeit bezieht sich ausschließlich auf den Programmteil, der den Algorithmus ausführt. Das Einlesen der Datei, die Eingabe und die Ausgabe sind nicht enthalten. Die Laufzeit wurde mit Python BuiltIn-Modul „time“ berechnet. Das Programm wurde auf einem Laptop mit dem CPU Intel Core i5-3317U ausgeführt. Dieser Prozessor hat 4 Kerne à 1.70GHz und erschien vor ca. 8 Jahren.

4.2 Beispiel 1

Hier ist das Beispiel „stromrallye0.txt“ abgebildet. Zur Veranschaulichung wird hier außerdem einmal die grafische Ausgabe präsentiert.

```
Weg gefunden! Koordinaten des Weges:  [[3, 5], [3, 4], [4, 4], [5, 4], [5, 3], [
5, 2], [5, 1], [5, 2], [5, 3], [5, 4], [5, 3], [4, 3], [3, 3], [3, 2], [2, 2], [
1, 2], [2, 2], [3, 2]]
================================================================================
Laufzeit: 0.00517725944519043
```



4.3 Beispiel 2

Hier wird die Ausgabe des Programms für die Datei „stromrallye3.txt“ abgebildet. Dieses Beispiel kann nicht gelöst werden.

```
Es wurde kein Weg gefunden.
0.008870601654052734
```

4.4 Beispiel 3

Hier wird die Ausgabe des Programms für die Datei „stromrallye4.txt“ abgebildet. Die grafische Ausgabe wird nicht mit abgebildet, da sie viel zu groß ist.

```
Weg gefunden! Koordinaten des Weges:  [[40, 25], [41, 25], [42, 25], [43, 25], [
44, 25], [45, 25], [46, 25], [47, 25], [48, 25], [49, 25], [50, 25], [51, 25], [
52, 25], [53, 25], [54, 25], [55, 25], [56, 25], [57, 25], [58, 25], [59, 25], [
60, 25]]
================================================================================
Laufzeit: 0.008121013641357422
```

4.5 Beispiel 4

Hier wird die Ausgabe des Programms für die Datei „stromrallye5.txt“ abgebildet. Dieses Beispiel kann ebenfalls nicht gelöst werden.

```
Es wurde kein Weg gefunden.
2.6185226440429688
```

4.6 Beispiel 5: Schwierigkeitsgrad Einfach

Hier wird ein von dem Programm aus der Teilaufgabe b) erstelltes Beispiel mit der Schwierigkeitsstufe „Einfach“ abgebildet, sowie die Lösung, die mit dem Hauptprogramm gefunden wurde.

```
Groesse des Spielfelds: 5  Anzahl der Batterien: 2
Batterieladung:2
*  *  *  *  *
*  5  *  *  *
*  1  *  X  *
*  *  *  *  *
*  *  *  *  *
```

```
Weg gefunden! Koordinaten des Weges:  [[4, 3], [3, 3], [2, 3], [2, 2], [3, 2], [
4, 2], [5, 2], [4, 2], [5, 2]]
================================================================================
Laufzeit: 0.0028600692749023438
```

4.7 Beispiel 6: Schwierigkeitsgrad Mittel

Hier wird ein von dem Programm aus der Teilaufgabe b) erstelltes Beispiel mit der Schwierigkeitsstufe „Mittel“ abgebildet, jedoch keine Lösung des Hauptprogramms, da die Laufzeit hier bereits zu hoch ist.

```
Groesse des Spielfelds: 10  Anzahl der Batterien: 12
Batterieladung:41
*  *  *  *  *  11 11 *  *  *
*  *  *  *  *  *  7  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  7
24 *  *  19 *  12 21 *  *  *
*  *  *  8  *  *  *  *  *  *
*  *  *  *  *  *  22 *  *  20
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  X  *  *  *  28
```

4.8 Beispiel 7: Schwierigkeitsgrad Schwer

Hier wird ein von dem Programm aus der Teilaufgabe b) erstelltes Beispiel mit der Schwierigkeitsstufe „Schwer“ abgebildet, auch wieder ohne Lösung des Hauptprogramms.

```
Groesse des Spielfelds: 20  Anzahl der Batterien: 53
Batterieladung:49
*  68 *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
45 *  *  *  *  *  *  *  *  *  10 34 *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
*  15 *  *  *  *  *  *  13 *  *  *  *  *  *  *  *  *  *  *
*  *  *  76 44 *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  78 *  *  *  *  *  10 *  *  38 *  *
*  *  *  *  *  61 *  *  *  *  55 *  *  *  *  *  12 *  *  *
*  *  *  *  16 *  53 *  *  *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  9  *  *  49 *  *  38 *  21
*  *  *  *  *  *  *  *  *  *  *  *  38 69 *  *  *  *  57 *
*  *  *  *  *  *  *  *  44 *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  *  *  44 *  *  *  *  *  *
X  *  14 *  *  *  *  *  *  17 *  *  *  *  *  *  *  *  *  *
37 *  *  *  *  *  *  *  *  *  *  *  36 *  *  *  *  *  *  *
*  *  22 *  *  *  *  *  *  *  *  *  *  *  39 *  *  *  *  *
*  *  *  *  *  56 *  17 *  *  7  *  *  *  *  *  *  *  21 *
*  *  *  66 16 *  *  *  *  *  *  26 *  *  41 *  *  44 *  *
*  23 *  *  35 43 *  88 *  *  *  *  16 60 *  *  *  69 *  *
*  *  *  48 *  *  *  *  *  84 *  *  *  *  *  20 *  *  *  *
*  *  30 *  *  *  *  *  *  *  *  *  *  23 *  *  *  *  *  56
```

4.9 Beispiel 8: Sonderfall 1

Hier wird ein Beispiel gezeigt, bei dem eine der beiden implementierten Abbruchbedingungen vor dem Spiel eintreten.

```
Batterieladung:7
*  *  *  *  1
1  *  *  *  *
*  *  *  *  *
*  *  *  *  3
*  *  X  *  *
```

```
Da es mehr als eine 1 gibt, die keinen Nachbarn hat, ist diese Spielsituation ni
cht möglich
```

4.10 Beispiel 9: Sonderfall 2

Hier wird die andere Implementierte Abbruchbedingung vor dem Spiel gezeigt.

```
Batterieladung:7
3  *  *  *  *  *  *  *  *  *
1  *  *  4  *  *  *  *  *  *
X  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  3
```

```
Da mindestens eine Ersatzbatterie nicht erreichbar ist, ist diese Spielsituation
 nicht lösbar
```

5 Quelltext

5.1 Variablenliste Teilaufgabe a)

	Variablenname	Variablentyp	Variablenbeschreibung
Globale Variablen:	groesse	integer	Seitenlänge des Spielfelds
	roboter	list	Koordinaten und Ladung des Roboters
	anzahl_batterien	integer	Anzahl der Ersatzbatterien
	koordinaten_eingelesen	list	Koordinaten und Ladung der Ersatzbatterien
	wege_dict	dictionary	Gespeicherte Nachfolgewege
	zeit	float	Zeitmessung
	letztekoordinaten	list	Die Koordinaten des letzten Teil des Weges, bei dem die übrig geblieben Ladung verbraucht wird
Lokale Variablen:			
anfangscheck()	eins_counter	integer	Zähler von Ersatzbatterien mit der Ladung 1, die keine Nachbarn haben
	obnachbarn	boolean	Beinhaltet ob die momentane Ersatzbatterie mit der Ladung 1 Nachbarn hat
oballeine()	erreichbar	list	List der Ersatzbatterien, die momentane Ersatzbatterie erreichen können
	abstand	float	Minimaler Abstand zwischen den beiden Ersatzbatterien
	abstand_start	float	Minimaler Abstand zum Startpunkt des Robotors
rekursiv()	lokliste	list	Beinhaltet die Koordinaten des bis jetzt zurückgelegten Weges
	weg_davor	list	Koordinatend des vorherigen Weges
	ablage_batterie	integer	Batterie, die am momentanen Punkt abgelegt wird
	startpunkt	list	Momentaner Standpunkt des Roboters
	alle_wege	list	Alle Wege, die vom momentanen Standpunkt zu den nächsten Ersatzbatterien gefunden wurden
	weg	list	Koordinaten des aktuell zu behandelten Weges
	mind_batterieverbrauch	integer	Minimaler Batterieverbrauch, der benötigt wird um den Weg zu
	neuer_startpunkt	list	Koordinaten und Ladung des nächsten Startpunkts
	ebene	integer	momentane Ebene der Rekursion
summeerreichtebatterien()	summe	integer	Anzahl der Batterien, fungiert als negativer Zähler
	ersatzbatterie	list	Einzelne Ersatzbatterien der Liste koordinaten
obwegfertig()	richtungen	list	Die möglichen Richtungen, in die der Robotor gehen kann
	stop	boolean	Variable, um Überfunktionen der Rekursion zu signalisieren, dass Weg gefunden wurde
bewegen()	stop	boolean	Variable, um Überfunktionen der Rekursion zu signalisieren, dass Weg gefunden wurde
	bewegen_liste	list	Koordinaten des bereits durchgangenen Weges
aufruf_a_stern()	alle_im_feld	list	Alle möglichen Koordinaten
	hindernisse	list	Die Koordinaten aller Hindernisse auf dem Feld
	ersatzbatterien	list	Alle Ersatzbatterien, die vom momentanen Standpunkt aus angelaufen werden sollen
	alle_wege	list	Alle Wege, die vom momentanen Standpunkt zu den nächsten Ersatzbatterien gefunden wurden
	endpunkt	list	Der Endpunkt der Suche nach einem Weg
	abstand	float	Minimaler Abstand zwischen Standpunkt und Endpunkt
	summe	integer	Summe aller auf dem Feld liegenden Ladungen
	ladung	integer	Momentane Ladung des Roboters
	weg	list	Gefundener Weg
summeersatzbatterien()	summe	integer	Zählt die noch verbleibenden Ladungen auf dem Feld
oberreichbar()	nachbarn	list	Nachbarn des momentanen Punktes
	daneben	integer	Zählt wie viele Nachbarn eines Punktes es gibt, die besetzt sind oder nicht mehr im Spielfeld sind
a_stern()	open_list	list	Liste mit allen noch zu prüfenden Koordinaten und ihren Attributen
	closed_list	list	Liste der bereits untersuchten Koordinaten
	standpunkt	list	Momentaner Standpunkt
	current	list	Die Koordinate, die momentan geprüft wird
	momentaner_wert	list	Die Koordinaten des „Parent“
	richtungen	list	Richtungen zu den Nachbarn des momentanen Punktes
	neighbours	list	Beinhaltet die Nachbarn des momentanen Punktes
	position	list	Koordinaten eines Nachbarn
	g	integer	Kosten des Weges vom Startpunkt bis zum jetzigen Punkt
	h	float	gemessene/geschätzte Kosten vom jetzigen Punkt bis zum Zielpunkt
	f	float	Summe von „g“ und „h“

5.2 Programmcode Teilaufgabe a)

```python
#Für grafische Mitverfolgung
try:
    import matplotlib.pyplot as plt
except:
    pass
#Für Zeitmessung
import time
import math
#Zum Entfernen von störenden Zeichen bei Einlesen der Datei
import re
#Zum Kopieren von Listen in Listen
from copy import deepcopy

#Pfad der Datei
file = "stromrallye" + input("Zahl") + ".txt"

#Einlesen der Daten
with open(file, "r") as file:
    groesse = int(file.readline())
    #regex um Zeichen wie \n auszuschliessen
    roboter = [int(x) for x in (re.findall(r'\d+', file.readline()))]
    anzahl_batterien = int(file.readline())
    koordinaten_eingelesen = []
    for i in range(anzahl_batterien):
        koordinaten_eingelesen.append([int(x) for x in (re.findall(r'\d+', file.readline()))])

#Attribut für jede Ersatzbatterie hinzufügen, wenn die Batterie im Laufe des Weges noch nicht erreicht wurde --> 0, wenn schon --> 1
for o in range(len(koordinaten_eingelesen)):
    koordinaten_eingelesen[o].append(0)


```

```python
#Überprüfen, ob eine Ersatzbatterie nicht erreichbar ist
def oballeine(roboter, koordinaten):
    for koord1 in koordinaten:
        erreichbar = []
        for koord2 in koordinaten:
            #Mindestabstand berechnen
            abstand = abs(koord2[0]- koord1[0]) + abs(koord2[1]- koord1[1])
            if abstand <= koord2[2] and abstand != 0:
                erreichbar.append(koord2)
        abstand_start = abs(roboter[0]- koord1[0]) + abs(roboter[1]- koord1[1])
        if abstand_start <= roboter[2]:
            erreichbar.append(roboter)
        if len(erreichbar) == 0:
            return 1


#Grundcheck am Anfang, teilweise kann direkt erkannt werden, dass eingelesene Spielsituation nicht möglich ist
def anfangscheck(roboter, groesse, koordinaten):
    #Anzahl der Ersatzbatterien mit Ladung 1 zählen, die keine Nachbarn haben
    eins_counter = 0
    for koordinate in koordinaten:
        if koordinate[2] == 1:
            obnachbarn = False
            for nachbar in [[koordinate[0] - 1, koordinate[1]], [koordinate[0] + 1, koordinate[1]], [koordinate[0], koordinate[1] - 1], [koordinate[0], koordinate[1] + 1]]:
                for koord in koordinaten:
                    if nachbar == [koord[0], koord[1]] or nachbar == [roboter[0], roboter[1]]:
                        obnachbarn = True
                        break
            if obnachbarn == False:
                eins_counter += 1
    #wenn mehr als eine 1 ohne Nachbar --> Spiel nicht möglich
    if eins_counter > 1:
        print("Da es mehr als eine 1 gibt, die keinen Nachbarn hat, ist diese Spielsituation nicht möglich")
        exit()
    #Wenn es eine Ersatzbatterie gibt, die nicht erreicht werden kann --> Spielabbruch
    if oballeine(roboter, koordinaten) == 1:
        print("Da eine Ersatzbatterie nicht erreichbar ist, ist diese Spielsituation nicht möglich")


#Berechnung der Summe der Ladungen aller Ersatzbatterien
def summersatzbatterien(koordinaten):
    summe = 0
    for ersatzbatterie in koordinaten:
        summe += ersatzbatterie[2]

    return summe
```

```python
#Berechnung der Summe der Ladungen aller Ersatzbatterien
def summersatzbatterien(koordinaten):
    summe = 0
    for ersatzbatterie in koordinaten:
        summe += ersatzbatterie[2]

    return summe

#Überprüfen, ob bereits alle Batterien erreicht wurden (durch am Anfang hinzugefügtes Attribut)
def summeerreichtebatterien(koordinaten):
    summe = anzahl_batterien
    for batterie in koordinaten:
        summe -= batterie[3]
    if summe == 0:
        for batterie in koordinaten:
            if batterie[2] != 0:
                if batterie[2] % 2 != 0:
                    return 0
        return 1

#Überprüfen ob übergebener Punkt erreichbar ist, also er nicht umschlossen von Ersatzbatterien bzw. der Wand ist
def oberreichbar(standpunkt, endpunkt, ersatzbatterien, grid):
    nachbarn = [[endpunkt[0] - 1, endpunkt[1]], [endpunkt[0] + 1, endpunkt[1]], [endpunkt[0], endpunkt[1] - 1], [endpunkt[0], endpunkt[1] + 1]]
    daneben = 0
    for nachbar in nachbarn:
        if nachbar[0] == standpunkt[0] and nachbar[1] == standpunkt[1]:
            break
        if nachbar in ersatzbatterien:
            daneben += 1
        if 0 < nachbar[0] < grid+1 and 0 < nachbar[1] < grid+1:
            pass
        else:
            daneben += 1
    #wenn sie 4 Nachbarn bzw. Wand hat --> Ersatzbatterie ist nicht erreichbar
    if daneben == 4:
        return 1
    else:
        return 0

#Algorithmus A*
def a_stern(standpunkt, endpunkt, ladung, grid, hindernisse, alle_im_feld):
    #Zielpunkt aus Hindernissen entfernen
    hindernisse.remove(endpunkt)
    #open- und closed-list deklarieren
    open_list = []
    closed_list = []

    #startwert zur open list hinzufügen [koordinaten, g, h, f, vorheriger]
    open_list.append([standpunkt, 0, 0, 0, None])

    #solange es Einträge in der openlist gibt
    while len(open_list) > 0:
        #current bestimmen --> in open list mit niedrigstem f wert
        for i in range(len(open_list)):
            if open_list[i][3] == min([item[3] for item in open_list]):
                current = open_list[i]
                break

        #current aus open löschen und zu closed hinzufügen
        open_list.remove(current)
        closed_list.append(current)

        #wenn current ziel ist --> ende
        if current[0] == endpunkt:
            #Weg zurückverfolgen, in dem immer der "Parent"(Vorgänger) von jedem Punkt genommen wird, bis der "Parent" None ist
            weg = []
            weg.append(current[0])
            momentaner_wert = current[0]
            while momentaner_wert != None:
                for i in range(len(closed_list)):
                    if closed_list[i][0] == momentaner_wert:
                        weg.append(closed_list[i][4])
                        momentaner_wert = closed_list[i][4]
            #reverse Liste und entfernen des Startpunktes
            weg = weg[::-1]
            weg = weg[1:]
            return weg

        #alle Richtungen der Nachbarn
        richtungen = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        neighbours = []

        #Nachbarn bestimmen und prüfen ob sie noch im Feld sind
        for richt in richtungen:
            position = [current[0][0] + richt[0], current[0][1] + richt[1]]
            #wenn Position im Feld
            if position in alle_im_feld:
                #und wenn Position nicht auf einem Hindernis(einer Ersatzbatterie)
                if position not in hindernisse:
                    neighbours.append(position)
```

```
        #Über Nachbarn iterieren
        for neighbour in neighbours:
            #Wenn Nachbar in closed list ist --> überspringen
            if neighbour in [i[0] for i in closed_list]:
                continue
            #g, h und f bestimmen
            #g ist +10 von dem davor
            g = current[1] + 1
            #h mit Satz des Pythagoras ohne Wurzel bestimmen
            h = math.sqrt(((neighbour[0] - endpunkt[0])**2 + (neighbour[1] - endpunkt[1])**2))
            #f ist die Summe aus g und h
            f = g + h

            #Wenn der Nachbar bereits in open list ist, und der g wert dieses mal größer ist --> continue
            for elem in open_list:
                if elem[0] == neighbour:
                    if g > elem[1]:
                        continue
                    else:
                        open_list.remove(elem)
            #Hinzufügen von Nachbar in open_list
            open_list.append([neighbour, g, h, f, current[0]])
    return

def aufruf_a_stern(standpunkt, ladung, groesse, koordinaten):
    #alle Koordinaten die im Feld sind
    alle_im_feld = []
    for i in range(1, groesse+1):
            for j in range(1, groesse+1):
                alle_im_feld.append([i, j])

    #die Koordinaten der Ersatzbatterien
    hindernisse = []
    ersatzbatterien = []
    for i in range(len(koordinaten)):
        hindernisse.append([koordinaten[i][0], koordinaten[i][1]])
        if [koordinaten[i][0], koordinaten[i][1]] != standpunkt:
            ersatzbatterien.append([koordinaten[i][0], koordinaten[i][1]])

    #Aufruf der Funktion a_stern für jede Ersatzbatterie
    alle_wege = []
    for endpunkt in ersatzbatterien:
        #Mindestabstand zwischen Startpunkt und Endpunkt ohne Hindernisse
        abstand = abs(endpunkt[0]- standpunkt[0]) + abs(endpunkt[1]- standpunkt[1])

        summe = summersatzbatterien(koordinaten)
        #Wenn der Abstand größer ist als die Summe aller Ersatzbatterien sowie der Ladung --> return, da eine Batterie nicht mehr erreichbar ist
        if abstand > summe + ladung:
            return 1
        #Wenn der Abstand größer ist als die Ladung --> continue mit nächster Batterie
        if abstand > ladung:
            continue
        #Wenn Funktion oberreichbar 0 zurückgibt (Batterie ist erreichbar) --> Aufruf der Funktion a_stern
        if oberreichbar(standpunkt, endpunkt, ersatzbatterien, groesse) == 0:
            weg = a_stern(standpunkt, endpunkt, ladung, groesse, hindernisse[:], alle_im_feld)
            #Hinzufügen von weg zu alle_wege
            alle_wege.append(weg)

    return alle_wege


#Rekursive Funktion "bewegen", um die letzte Ladung, die noch übrig geblieben ist und nirgenwo hinführt zu verbrauchen
def bewegen(startpunkt, koordinaten, richtung, richtungen, bewegen_liste):
    #Abbruchbedingung
    global stop
    if stop == True:
        return
    #Wenn die Ladung auf 0 ist --> stop
    if startpunkt[2] == 0:
        global letztekoordinaten
        letztekoordinaten = bewegen_liste
        stop = True
        return

    #Überprüfen ob Roboter auf diesen Punkt gehen kann
    x = startpunkt[0] + richtung[0]
    y = startpunkt[1] + richtung[1]
    if 0 < x <= groesse and 0 < y <= groesse:
        if [x,y] not in [[i[0],i[1]] for i in koordinaten_eingelesen]:
            #Richtung auf Koordinaten des Roboters addieren
            startpunkt[0] += richtung[0]
            startpunkt[1] += richtung[1]
            #Ladung um 1 verringern
            startpunkt[2] -= 1
        else:
            return
    else:
        return
    #Neue Koordinaten zu lokaler Liste hinzufügen um am Ende ein Ergbenis zu haben
    bewegen_liste.append([startpunkt[0], startpunkt[1]])

    #Funktion rekursiv aufrufen mit allen 4 Richtungen
    for elem in richtungen:
        bewegen(startpunkt[:], deepcopy(koordinaten), elem, richtungen[:], deepcopy(bewegen_liste))
```

```python
#Überprüfen ob gefundener Weg geeignet ist
def obwegfertig(startpunkt, koordinaten, lokliste):
    #Weg für die letzte Ladung finden
    richtungen = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    global stop
    stop = False
    global letztekoordinaten
    letztekoordinaten = []
    for elem in richtungen:
        bewegen(startpunkt[:], deepcopy(koordinaten), elem, richtungen[:], [])
    #Wenn nicht gefunden --> weiter nach einem Weg suchen
    if letztekoordinaten == []:
        return 0

    #Überprüfen ob übrig gebliebene Ladungen entfernt werden können
    else:
        ersatzbatterien = [[i[0], i[1]] for i in koordinaten]
        for koord in koordinaten:
            if koord[2] > 0:
                suchkoordinate = [koord[0], koord[1]]
                for index, elem in reversed(list(enumerate(lokliste))):
                    if elem == suchkoordinate:
                        #Überprüfen ob bei einem Punkt, an dem Ladung übrig geblieben ist, der Weg davor mindestens die Länge von 2 hatte
                        if abs(index-2) == index-2:
                            if lokliste[index - 1] in ersatzbatterien or lokliste[index-2] in ersatzbatterien:

                                return 0
                            #Die beiden Punkte vor Erreichen des Weges kopieren und so oft in Weg einfügen, sodass Ladung aufgebraucht ist
                            else:
                                davor = lokliste[index - 1]
                                nochmaldavor = lokliste[index - 2]
                                for i in range(koord[2]//2):
                                    lokliste.insert(index, nochmaldavor)
                                    lokliste.insert(index + 1, davor)
                        else:
                            davor = lokliste[index-1]
                            nochmaldavor = [roboter[0],roboter[1]]
                            for i in range(koord[2]//2):
                                    lokliste.insert(index, nochmaldavor)
                                    lokliste.insert(index + 1, davor)
    return lokliste

#Hauptfunktion rekursiv
def rekursiv(startpunkt, groesse, koordinaten, ablage_batterie, lokliste, weg_davor, ebene):
    #Hinzufügen des vorherigen Weges zur lokalen Liste lokliste, in der am Ende der komplette Weg gespeichert ist
    lokliste = lokliste + weg_davor[1:]

    #Aktuellen Standort in der Liste "koordinaten" finden, um die Ablagebatterie dort abzulegen
    #Außerdem wird die Batterie als "bereits erreicht" markiert
    for j in range(len(koordinaten)):
        if koordinaten[j][0] == startpunkt[0] and koordinaten[j][1] == startpunkt[1]:
            koordinaten[j][2] = ablage_batterie
            koordinaten[j][3] = 1

    #Wenn alle Batterien als "bereits erreicht" markiert wurden
    if summeerreichtebatterien(koordinaten) == 1:
        #Wenn Funktion obwegfertig() nicht 0 zurückgibt, d.h. Weg geeignet ist --> Programm beenden und Weg sowie Laufzeit ausgeben
        #Danach Programm beenden
        obfertig = obwegfertig(startpunkt, deepcopy(koordinaten), deepcopy(lokliste))
        if obfertig != 0:
            ausgabeliste = obfertig + letztekoordinaten
            ausgabeliste.insert(0, [roboter[0], roboter[1]])
            print("Weg gefunden! Koordinaten des Weges: ", ausgabeliste)
            print(80*"=")
            print("Laufzeit:", time.time() - zeit)
            grafische = input("Grafische Ausgabe erwünscht?[J/n]")
            if grafische == "J" or grafische == "j":
                grafisch(ausgabeliste)
            exit()
        #Wenn Weg nicht geeignet ist --> weitersuchen
        else:
            pass

    #Wenn der momentane Startpunkte bereits in wege_dict gespeichert ist, d.h. alle Wege für diesen Punkt bereits berechnet wurden --> Wege übernehmen
    if str(startpunkt) in wege_dict:
        alle_wege = wege_dict[str(startpunkt)]
    #Sonst alle Wege suchen
    else:
        alle_wege = aufruf_a_stern([startpunkt[0], startpunkt[1]], startpunkt[2], groesse, koordinaten[:])
        #Wenn kein Weg gefunden wurde
        if alle_wege == 1:
            return
    #print(alle_wege)
    #Alle Wege zum Dictionary hinzufügen
    wege_dict[str(startpunkt)] = alle_wege

    #Wahlweise Funktion um Suchen grafisch zu verfolgen
    #matplot(koordinaten, roboter, lokliste, groesse)

    #Für jeden Weg in alle_wege
    for weg in alle_wege:
        #Wenn der Weg "None" ist --> überspringen
        if weg == None:
            continue
        #Minimalen Batterieverbrauch berechnen
        mind_batterieverbrauch = len(weg) - 1
        #Wenn der minimale Batterieverbrauch größer als die momentane Ladung ist --> continue
        if mind_batterieverbrauch > startpunkt[2]:
            continue
```

```
        #Neuen Startpunkt bestimmen
        for l in range(len(koordinaten)):
            if koordinaten[l][0] == weg[-1][0] and koordinaten[l][1] == weg[-1][1]:
                neuer_startpunkt = koordinaten[l]
        #Wenn der neue Startpunkt eine Ladung von 0 hat --> mit nächstem Weg weitermachen
        if neuer_startpunkt[2] == 0:
            continue

        #rekursiver Aufruf der Funktion
        rekursiv(neuer_startpunkt[:], groesse, deepcopy(koordinaten), startpunkt[2] - mind_batterieverbrauch, deepcopy(lokliste), deepcopy(weg), ebene + 1)


def main():
    global wege_dict
    wege_dict = {}
    #Zeitstart
    global zeit
    zeit = time.time()
    #Grundüberprüfung, ist eingelesene Spielsituation möglich?
    anfangscheck(roboter, groesse, koordinaten_eingelesen)
    #Aufruf der Rekursion
    rekursiv(roboter[:], groesse, deepcopy(koordinaten_eingelesen), 0, [], [], 0)

    #Wenn kein Weg gefunden wurde
    print("Es wurde kein Weg gefunden.")
    print(time.time() - zeit)

main()
```

5.3 Programmcode Teilaufgabe b)

```
def wegfinden(startpunkt, alle_im_feld, groesse):
    while True:
        #Random naechste Batterie aus ersatzbatterien nehmen
        naechste_batterie = choice(ersatzbatterien)
        if naechste_batterie == startpunkt:
            continue
        #Hindernisse auflisten
        hindernisse = []
        for i in range(len(ersatzbatterien)):
            if [ersatzbatterien[i][0], ersatzbatterien[i][1]] != naechste_batterie[:-1]:
                hindernisse.append([ersatzbatterien[i][0], ersatzbatterien[i][1]])
            else:
                #"bereits erreicht" Index auf 1 setzen
                ersatzbatterien[i][2] = 1

        #schnellsten Weg suchen
        weg = a_stern(startpunkt, [naechste_batterie[0], naechste_batterie[1]], groesse, hindernisse, alle_im_feld)
        #wenn nicht gefunden, anderen naechsten Punkt nehmen
        if weg == None:
            continue
        #Weg in Liste speichern
        global alle_wege
        alle_wege.append([startpunkt, naechste_batterie, len(weg)-1])
        return naechste_batterie

#Berechnet ob bereits alle Batterien mindestens einmal erreicht wurden
def summeerreichtebatterien(anzahl_batterien):
    summe = anzahl_batterien
    for batterie in ersatzbatterien:
        summe -= batterie[2]
    if summe == 0:
        return 1
    return
```

```python
#Berechnet die Ladungen der Ersatzbatterien
def ladungen():
    koordinaten = []
    neue_ladung = 0
    for i in range(len(alle_wege)):
        changed = False
        for j in range(len(koordinaten)):
            #wenn Ersatzbatterie bereits mindestens einmal im Weg erreicht wurde
            if [koordinaten[j][0], koordinaten[j][1]] == [alle_wege[i][1][0], alle_wege[i][1][1]]:
                changed = True
                #neue Ladung für den nächsten Weg mitnehmen
                temp_neue_ladung = koordinaten[j][2]
                if neue_ladung:
                    koordinaten[j][2] =  alle_wege[i][2]+neue_ladung
                    neue_ladung = 0
                else:
                    #aktuelle Ladung speichern
                    koordinaten[j][2] =  alle_wege[i][2]
                neue_ladung = temp_neue_ladung
                break
        #wenn Ersatzbatterie bereits geändert wurde --> weiter mit der nächsten
        if changed == True:
            continue
        #sonst neue Ersatzbatterie in Liste koordinaten[] anlegen
        else:
            if neue_ladung:
                koordinaten.append([alle_wege[i][1][0], alle_wege[i][1][1], alle_wege[i][2]+neue_ladung])
                neue_ladung = 0
            else:
                koordinaten.append([alle_wege[i][1][0], alle_wege[i][1][1], alle_wege[i][2]])
    return koordinaten

def main():
    #Attribute des Schwierigkeitsgrades
    global groesse
    if schwierigkeitsgrad == 0:
        groesse = 5
        anzahl_batterien = randint(2,5) + 1
    elif schwierigkeitsgrad == 1:
        groesse = 10
        anzahl_batterien = randint(10, 15) + 1
    elif schwierigkeitsgrad == 2:
        groesse = 20
        anzahl_batterien = randint(40, 60) + 1

    #Startpunkt ermitteln
    startpunkt = [randint(1, groesse), randint(1, groesse)]
    print("Groesse des Spielfelds:", groesse, " Anzahl der Batterien:", anzahl_batterien-1)
    global ersatzbatterien
    ersatzbatterien = []
    global alle_wege
    alle_wege = []
    #alle Koordinaten im Feld in Liste speichern
    alle_im_feld = []
    for i in range(1, groesse+1):
            for j in range(1, groesse+1):
                alle_im_feld.append([i, j])

    #random Ersatzbatterien im Feld platzieren
    i = 0
    while i < anzahl_batterien:

        x = randint(1, groesse)
        y = randint(1, groesse)
        if [x,y,0] not in ersatzbatterien and [x,y] != startpunkt:
            ersatzbatterien.append([x,y,0])
            i += 1
    while True:
        startpunkt = wegfinden(startpunkt, alle_im_feld, groesse)
        if summeerreichtebatterien(anzahl_batterien) == 1:
            return
```