



Bundesweite-Informatikwettbewerbe [3]

GYMNASIUM MAINZ-OBERSTADT

FACHARBEIT IM LEISTUNGSKURS INFORMATIK

Bundeswettbewerb Informatik 38

Tobias Bück

Am Judensand 7, 55122 Mainz

betreut von

Jörg SCHAEDE

Diese Arbeit handelt von der Bearbeitung des Bundeswettbewerb Informatik 38. Dabei werden Graphentheorie, Algorithmen und Datenstrukturen zur Lösung komplexer Probleme genutzt. In der ersten Aufgabe wird ein Computer-Programm entwickelt, welches das Brettspiel Stromrallye löst. Bei der 2. Aufgabe geht es darum in einem Straßennetz den Weg zu finden, der am schnellsten ist, aber auch wenige Abbiegungen beinhaltet. [2]

1 Inhaltsverzeichnis

Inhaltsverzeichnis

1	Inhaltsverzeichnis	2
2	Einleitung	4
3	Hauptteil	4
3.1	Aufgabe 1	4
3.1.1	Aufgabenstellung	4
3.1.2	Lösungsidee	4
3.1.3	Kürzester Weg	4
3.1.4	Erklärung Graph	6
3.1.5	Breiten Suche	7
3.1.6	Djikstra	7
3.1.7	A-Stern	8
3.1.8	Kürzester Weg-Fazit	8
3.1.9	Kürzester Weg - Laufzeitanalyse	8
3.1.10	Sonderfälle	8
3.1.11	Wege zu Ersatzbatterien	9
3.1.12	Speicherung der Möglichkeiten	9
3.1.13	Erklärung - Baum	10
3.1.14	Lösung finden	10
3.1.15	Insgesamte Laufzeitanalyse	11
3.1.16	Umsetzung	11
3.1.17	UML-Klassen-Diagramm	13
3.1.18	Lösungen	14
3.2	Aufgabe 1 b)	14
3.2.1	Beschreibung	14
3.2.2	Lösungsidee	14
3.2.3	Schwierigkeit des Spiels	15

3.2.4	Lösung	15
3.2.5	Umsetzung	15
3.3	Aufgabe 3	15
3.3.1	Aufgabenstellung	15
3.3.2	Lösungsidee	15
3.3.3	Abbiegungen bestimmen	17
3.3.4	HashSet	18
3.3.5	HashMap	18
4	Fazit	18
5	Anhang	18
5.1	Offizielle Aufgabenbeschreibung des BWinf	18
5.2	Programm Erklärung Aufgabe 1	21
5.2.1	SolutionTree	21
5.2.2	Node	21
5.2.3	Edge	22
5.2.4	Entity	22
5.2.5	Position	23
5.2.6	Roboter	24
5.2.7	Enum-Directions	25
5.2.8	SpareBatterie	25
5.2.9	Spielfeld	25
5.2.10	Graph	26
5.2.11	Vertex	27
5.3	Umsetzung Aufgabe 1 b)	27
5.4	Beispiel3	28
5.4.1	Eingabe	28
5.4.2	Lösung	28
5.5	Beispiel4	29
5.5.1	Eingabe	29
5.5.2	Lösung	29
6	Quellen	30
7	Erklärung über die selbständige Anfertigung der Arbeit	31

2 Einleitung

Beim Bundeswettbewerb Informatik 38 habe ich mich für die Bearbeitung der Aufgaben 1 und 3 entschieden, da mir diese am interessantesten erschienen. Der Bundeswettbewerb Informatik ist ein Wettbewerb für junge Programmierer aus Deutschland. Der Wettkampf findet in Runden statt, bei dieser Arbeit geht es um die Bearbeitung der 2. Runde.

In der 2. Runde werden 2 von 3 möglichen Aufgaben gelöst. Die Aufgaben sind häufig etwas kniffliger zu lösen und das Programmieren zum Lösen der Aufgaben spielt auch eine entscheidende Rolle.

Die ausführliche Dokumentation der Aufgaben ist beim BWinf (Bundeswettbewerb Informatik) essentiell wichtig, um in die nächste Runde zu kommen.[1]

3 Hauptteil

3.1 Aufgabe 1

3.1.1 Aufgabenstellung

Die Aufgabenstellung befindet sich im Anhang (siehe Kapitel 5.1 auf Seite 18). *Bundesweite Informatikwettbewerbe [2]*

3.1.2 Lösungsidee

Damit das Ziel, die Ladung aller Batterien zu verbrauchen, erreicht werden kann muss der Roboter sich zu den Batterien bewegen. Zu jeder Batterie mit einem positiven Ladestand muss der Roboter bewegt werden. Dabei gibt es aber mehrere Batterien, zu denen der Roboter gehen kann. Von diesen Batterien kann der Roboter wieder zu weiteren Batterien laufen.

3.1.3 Kürzester Weg

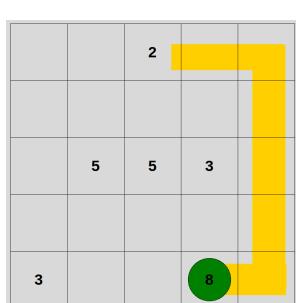


Abb. 1: Kürzester Weg

In Abbildung 1 sieht man den kürzesten Weg vom Roboter zur Ersatz-Batterie mit der Ladung 2. Der kürzeste Weg ist gelb gekennzeichnet. Der Roboter kann nicht den direkten Weg gehen, da die Ersatz-Batterien 5 und 3 den Weg versperren.

Manchmal ist es sinnvoll nicht den **kürzesten Weg** zu einer anderen Batterie zu gehen. Es kann auch vorteilhaft sein einen längeren Weg zu laufen. Dabei kann man fast alle Wege um eine **gerade Zahl verlängern**, indem man hin und her läuft, eine Verlängerung um eine ungerade Zahl ist im Umkehrschluss

dagegen unmöglich. Wenn man einen Weg vom Roboter zu einer Ersatz-Batterie finden will, dann sind alle anderen Ersatz-Batterien Hindernisse, welche nicht begehbar sind.

Weg Verlängern In Abbildung 2 wird gezeigt, wie ein Weg verlängert werden kann.

Stromrallye Darstellungsform Die kleinen Zahlen im oberen Bereich der Spielfelder zeigen die Lösungsschritte. Eine Zahl x bedeutet, dass dieses Spielfeld beim x ten Schritt vom Roboter besucht wird. Felder, welche keine Zahlen haben, werden nie besucht.

Diese Darstellungsform habe ich gewählt, da diese noch bei sehr komplexen Spielen

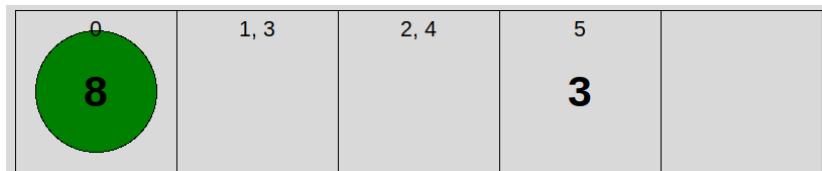


Abbildung 2: Weg Verlangern

leicht verständlich ist und es einfach war, ein Programm zu schreiben, welches diese generiert. In diesem Beispiel ist der kürzeste Weg 3 Schritte lang, wurde aber auf 5 Schritte verlängert. Der Weg kann auch weiter verlängert werden, indem noch öfter hin und her gelaufen wird.

Die **Kürzesten Wege** vom Roboter zu den anderen Batterien werden durch die **Breiten Suche** [9] herausgefunden, dies ist **A-Stern** [8] vorzuziehen, da dem Algorithmus A-Stern nur den Weg von einem Knoten zu einem anderen berechnet, die Breiten Suche findet bei einer Durchführung den kürzesten Weg zu allen anderen Knoten. Auch der **Djikstra** [7] Algorithmus findet nach einer Durchführung den kürzesten Weg zu allen Knoten. **Breiten Suche** ist trotzdem die beste Wahl, da **Breiten Suche** die **Zeitkomplexität**, $E = \text{Kanten}; V = \text{Knoten}$ $O(E+V)$ und **Djikstra** $O(V*\log(V)+E)$ (wenn Djikstra mit Fibonacci-Heap verwendet wird) hat. Somit ist die **Breiten Suche schneller als Djikstra**.

Der Graph Man kann das Spielfeld in einen Graphen umwandeln, dabei sind die Spielfelder die Knoten.

Es handelt sich um einen ungerichteten, ungewichteten Graphen, alle Konten sind in beide Richtungen begehbar und haben das Gewicht 1.

Wenn man versucht den kürzesten Weg vom Roboter zu einer Batterie zu finden, sind alle anderen Batterien Hindernisse, da man eine Batterie aufnehmen muss, wenn man auf ihr Feld geht. Deswegen sind im erzeugten Graph alle

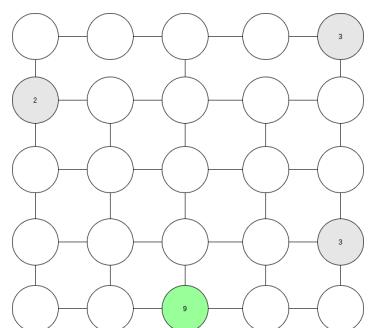


Abb. 3: Das Stromrallye Spiel als Graph

Felder mit Batterien, außer der Ziel-Batterie, nicht begehbar.

In Abbildung 3 sind zum besseren Verständnis die Ersatzbatterien in grau gekennzeichnet, der Roboter in grün.

In unserem Graphen, wird nicht gespeichert, wie groß die Ladung der Ersatzbatterien ist, dies ist in der Grafik nur für ein besseres Verständnis so dargestellt. Der Graph speichert aber, ob ein Feld (dies entspricht im Graphen einem Knoten) begehbar ist. Ersatzbatterien sind nicht begehbar, wenn man den Weg zu einer anderen Batterie finden will.

3.1.4 Erklärung Graph

Ein Graph ist eine Datenstruktur und besteht aus Knoten und Kanten, wobei die Knoten mit den Kanten verbunden sind ([biggs1986graph](#)). Es werden gerichtete und ungerichtete Graphen unterschieden. In gerichteten Graphen haben Kanten eine Richtung, in ungerichteten nicht. Man kann von einem Knoten über eine Kante zu einem anderen Knoten kommen usw.. In einem gewichteten Graphen haben Kanten zusätzlich ein Gewicht. Dies bedeutet, das ein bestimmtes Gewicht (oder Kosten) verwendet wird, um von dem einen Knoten zum anderen zu kommen.[12]

Beispielsweise in einem U-Bahn Netz stehen die Stationen für die Knoten. Die Kanten sind U-Bahn-Strecken und die Kosten der Kanten geben die Fahrtzeit der Strecken an.

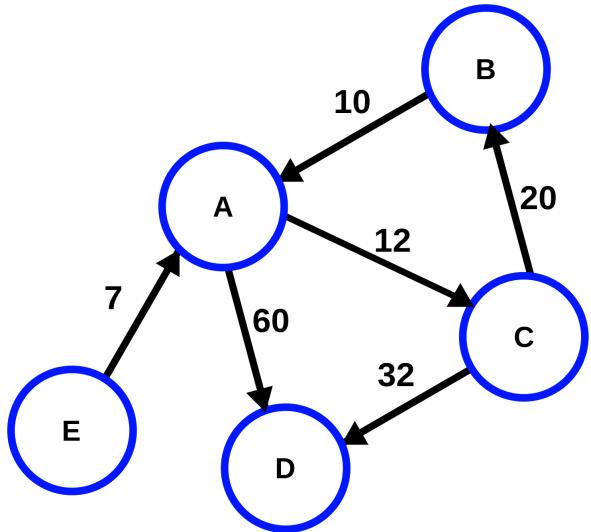


Abbildung 4: Ein Graph wikipedia [5]



Abbildung 5: Wiener U-bahn Netz wikipedia [6]

3.1.5 Breiten Suche

Bei der Breiten Suche startet man vom Startknoten und überprüft dann alle Kinder, bzw. Nachbarn des Startknotens, ob sie der Zielknoten (oder eines der Zielknoten) sind. Dies wird veranschaulicht in der Abbildung . Es folgt eine rekursive Durchführung für jeden Knoten. In einem Set (Erklärung in Kapitel 3.3.4) wird gespeichert, welche Knoten bereits besucht worden sind und deswegen nicht nochmal besucht werden müssen.[9]

Der Algorithmus terminiert, wenn entweder Wege zu allen Zielknoten gefunden oder alle Felder besucht worden sind (es kann sein, dass es zu einem Zielknoten gar keinen Weg gibt).

Die Breiten Suche funktioniert nur in einem Graph ohne Kanten Kosten, bzw. alle Kanten Kosten müssen gleich sein. In unserem Fall ist diese Bedingung gegeben, deswegen können wir die Breiten Suche verwenden, um den kürzesten Weg zu finden. In unserem Graph hat jede Kante den Kosten 1, da der Roboter bei jedem Schritt eine Energieladung verbraucht.

In unserem Fall kann die Breiten Suche auch bereits dann abgeschlossen werden, wenn alle Felder in der Reichweite des Roboters besucht wurden.

In Abbildung 6 sieht man eine durchgeführte Breiten Suche für das Stromrallye Spiel. Es werden die kürzesten Wege zu allen Feldern berechnet. Der kürzeste Weg zu einem Feld wird in der Abbildung, wie auch im Programm, gefunden, indem man vom Ziel-Feld zum Start geht und zwar so, dass die Farbe der abgelaufenen Felder sich von rot->leuchtrot->dunkel orange -> orange -> gelb -> grün verändert.

Die Anzahl der Schritte zu einem Feld sind:

- 1 Schritt: grün
- 2 Schritte: gelb
- 3 Schritte: orange
- 4 Schritte: dunkel orange
- 5 Schritte: Leuchtrot
- 6 Schritte: Rot

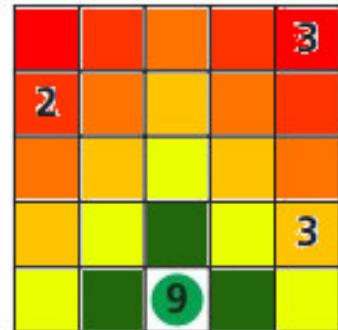


Abbildung 6: Spielfeld

<https://www.ralfarbpalette.de/ral-classic/ral-3024-leuchttrot> , <https://www.99colors.net/color-names>

3.1.6 Djikstra

Der Djikstra Algorithmus funktioniert ähnlich wie die Breiten Suche. Zusätzlich sortiert der Djikstra Algorithmus die Knoten nach Kosten und besucht zuerst den Knoten, zu dem ein Weg mit geringsten Kosten gefunden wurde. [7]

Dies bringt in unserem Fall nichts, da alle Knoten die gleichen Kosten haben und zwar

- Das Sortieren der Knoten macht allerdings den Djikstra Algorithmus langsamer als die Breiten Suche.

3.1.7 A-Stern

A-Stern nutzt eine zusätzliche Heuristik , um den bestmöglichen Pfad zuerst zu verwenden. In unserem Fall wäre so eine Heuristik, wie weit der Knoten vom Zielknoten entfernt ist. Im A Stern Algorithmus werden dann Knoten, die näher am Zielknoten sind, zuerst besucht. Dies macht den A Stern Algorithmus deutlich schneller als Djikstra und Breiten Suche. Die Breiten Suche kann man durch die Verwendung einer Heuristik nochmal stark verbessern. Da die Wahrscheinlichkeit, dass der schnellste Weg in der Richtung des Zielorts liegt, viel höher ist, wird dies bei A Stern clever genutzt, um den Algorithmus wesentlich schneller zu machen. [8]

3.1.8 Kürzester Weg-Fazit

Die Breiten Suche ist am schnellsten, da diese mit einer Durchführung die kürzesten Wege zu allen Ziel-Knoten findet und außerdem keine zusätzliche Zeit durch Sortieren der Knoten verbraucht.

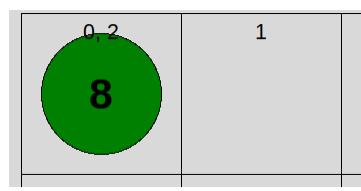
3.1.9 Kürzester Weg - Laufzeitanalyse

In unserem ungerichteten Graphen ist der Weg von A nach B genauso lang und hat genau die gleichen Stationen (in umgekehrter Reihenfolge) wie der Weg von B nach A. Damit die Breiten Suche nicht immer wieder neu ausgeführt wird, speichern wir unsere vorherigen Ergebnisse in einer HashMap. Der kürzeste Weg wird immer vom Roboter zu den Ersatzbatterien berechnet und er wird nur dann berechnet, wenn der Roboter auf einer Ersatzbatterie steht oder bei der Startposition. Es gibt so viele Zielorte wie die **Anzahl-Ersatzbatterien**, außer wenn der Roboter sich beim Start schon auf einer Ersatzbatterie befindet, dann einer weniger. Also wenn e die Anzahl der Ersatzbatterien ist, dann werden maximal $\sum_{n=0}^e n$ viele Wege berechnet. Bei der Ausführung von Breiten Suche werden direkt von einem Start Wege zu allen Zielen gefunden. Deswegen muss die Breiten Suche maximal $e - 1$ oft ausgeführt werden. Da die Breiten Suche $O(E + V)$ lange dauert, kommen wir so auf eine worst-case Laufzeitkomplexität von $O(e * (E + V))$.

3.1.10 Sonderfälle

Von einer Ersatzbatterie kann es auch Sinn machen wieder zu sich selbst zu laufen, wobei der Weg zu sich selbst immer 2 Schritte lang ist. Es kann aber auch sein, dass es keinen

Weg zu sich selbst gibt.
Abb. 7: Weg zu sich selbst



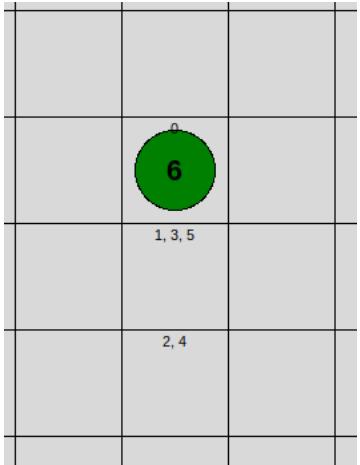


Abb. 8: Weg ins nichts!
Benutzung der ganzen Batterie-Ladung

Wenn alle Batterie-Ladungen 0 sind, nur der Roboter noch Ladung hat, muss dieser seine Ladung verlieren, indem er weitere Schritte läuft. Es kann aber vorkommen, dass dies nicht funktioniert, da alle Wege von anderen Batterien versperrt sind. Deswegen muss dies überprüft werden.

3.1.11 Wege zu Ersatzbatterien

Der **Roboter** kann von einer Position mit einem bestimmten Ladestand unterschiedliche Ersatzbatterien erreichen. Nachdem der Roboter zu einer dieser Ersatzbatterien geläufen ist, kann dieser von dort wieder zu verschiedenen Ersatzbatterien laufen.

Wie bereits beschrieben, kann es aber auch Sinn machen, dass der Roboter nicht den kürzesten Weg zu einer Ersatzbatterie geht. Da ein Weg immer um eine gerade Zahl verlängert werden kann, müssen diese Möglichkeiten auch durchgegangen werden. Also wenn der kürzeste Weg w lang ist und der Roboter b Batterieladung hat. Die Reihe a beinhaltet, alle möglichen Weglängen.

$$a_1 = w$$

$$a_{n+1} = x_n + 2\{n \in \mathbb{N} | w + 2n < b\}$$

Zum Beispiel: Wenn der Roboter mindestens 5 Schritte zu einer Batterie braucht und gerade 9 Ladung hat, dann sind 5, 7, 9 mögliche Anzahl Schritte um zur Batterie zu kommen.

Der kürzeste Weg ist 5 Schritte lang und der Roboter hat die Batterie-Ladung 9. Da der Weg um eine gerade Zahl verlängert werden kann, sind auch 7 und 9 möglich.

3.1.12 Speicherung der Möglichkeiten

Diese **Möglichkeiten** des Roboters werden in einem **Baum** gespeichert. Dabei werden in den **Knoten** der Spielstand (Position und Ladestand der Ersatzbatterien und des Roboters) und in den **Kanten** die Schritte und Anzahl an Schritten, welche der Roboter laufen muss, um den Spielstand a zu Spielstand b zu verändern, angegeben. Dabei ist Spielstand b auch wieder ein Spiel welches gelöst werden kann.

Um von Spielstand a zu Spielstand b zu kommen, muss der Roboter eine Schrittabfolge laufen. Der Lösungsspielstand, besteht aus einem leeren Roboter und leeren Batterien.

3.1.13 Erklärung - Baum

Ein Baum ist eine Art von Graph in welchem ein Knoten Kinder hat. Diese Kinder sind untereinander nicht verbunden. Knoten haben immer nur einen Eltern-Knoten. Der Eltern-Knoten muss eine Ebene oben drüber sein. Die Kinder eines Knotens sind immer eine Ebene unten drunter.

3.1.14 Lösung finden

Um nun herauszufinden, ob es eine Lösung gibt und welche diese ist, wird der Weg mit der größten Anzahl an Schritten gesucht. Wenn die Anzahl der Schritte der Summe der Batterieladungen (der Ersatzbatterien und des Roboters) entspricht, ist dies eine korrekte Lösung um den Ladestand aller Batterien auf 0 zu bringen. Abbildung: Lösung mit grün markiert

- RU : Batterie rechts unten auf dem Spiel
- RO: Batterie rechts oben auf dem Spiel
- LO: Batterie links oben
- RS: Roboters Startposition

3.1.15 Insgesamte Laufzeitanalyse

Größen sind:

- Größe des Spielfelds
- Anzahl Batterien
- Summe der Ladungen an Batterien

Die Laufzeit meines Algorithmus ist von all diesen genannten Größen abhängig. Je größer das Spielfeld, je mehr Batterien und je mehr Ladung, desto länger braucht mein Programm. Desto mehr Batterien, desto länger braucht der Algorithmus. Desto mehr Ladung, desto länger braucht mein Programm.

Am abhängigsten ist mein Algorithmus, aber von der Anzahl an Batterien, da sich dadurch der Möglichkeiten-Baum sehr stark vergrößert, weil dann die Zahl an Knoten zunimmt.

Der worst-case für meinen Algorithmus ist, dass sich auf jedem Feld des Spielfelds eine Batterie befindet, mit der Ladung 1. Dies führt zu einer Zeitkomplexität von $O(4^L)$; wenn L die Gesamt Summe der Batterien ist, welche dann äquivalent zur Anzahl Felder ist, also der quadrierten Größe. Daher gilt auch $O(4^{(s*s)})$; s für size entspricht der Größe des Spielfelds. Die 4, weil es 4 Richtungen gibt, in welche sich der Roboter bewegen kann.

Die Zeit für das Berechnen der kürzesten Wege fällt bei Spielen mit vielen Batterien nicht so stark ins Gewicht.

Die Breite Suche braucht $O(b*(E+V))$, b Anzahl Ersatzbatterien, E Edges, V gleich Vertices. Jeder Knoten hat maximal 4 Kanten (bzw. Nachbarn), also $O(E) = O(V)$, $O(b * (E + V)) = O(b * V)$. In unserem worst-case Szenario, beschrieben im vorherigen Absatz, entspricht die Anzahl der Knoten, der Anzahl an Feldern, also $b = s * s$; s für size ist die Größe des Spielfelds. V wiederum entspricht ebenfalls $s * s$. Daher kommen wir insgesamt auf $O(s * s * s * s)$, also $O(s^4)$. Da $b = s * s$. Können wir auch sagen $O(b^2)$. Also ist die Laufzeit der Breiten Suche quadratisch zur Anzahl Batterien. Die Laufzeit für den Möglichkeiten-Baum ist wesentlich größer. Sie ist $O(4^{(s*s)})$ oder $O(4^b)$. Beide Laufzeiten addiert: $O(4^b) + O(b^2) = O(4^b)$

Die insgesamte worst-case Laufzeit ist:

- In Abhängigkeit der Anzahl Batterien(b) => $O(4^b)$
- In Abhängigkeit der Größe des Spielfelds(s) => $O(4^{(s*s)})$
- In Abhängigkeit der gesamten Ladung(L) => $O(4^L)$

3.1.16 Umsetzung

Der Algorithmus wird in Python (Version 3.7) implementiert.

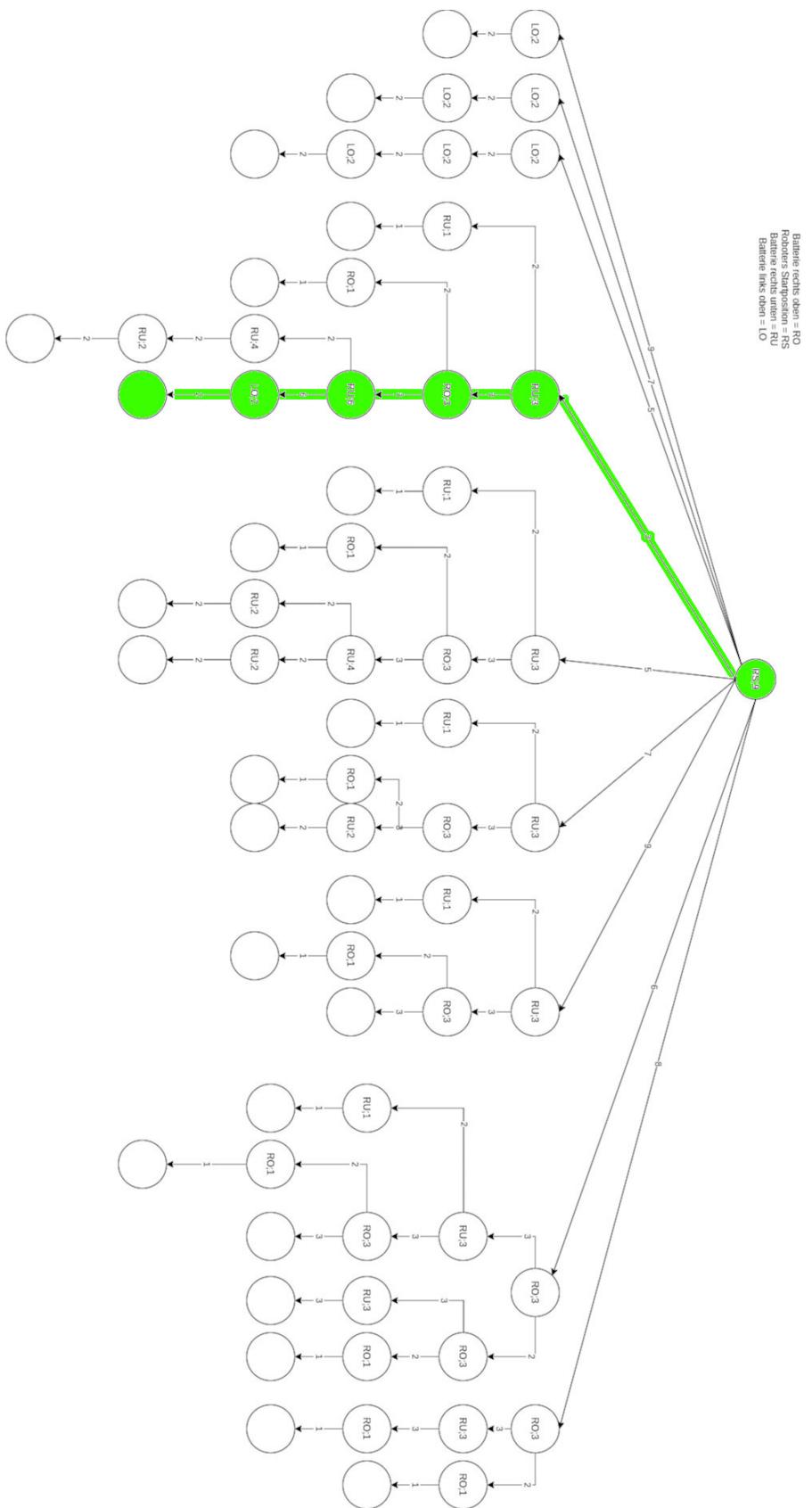


Abbildung 9: Schaubild des Möglichkeiten-Baums

3.1.17 UML-Klassen-Diagramm

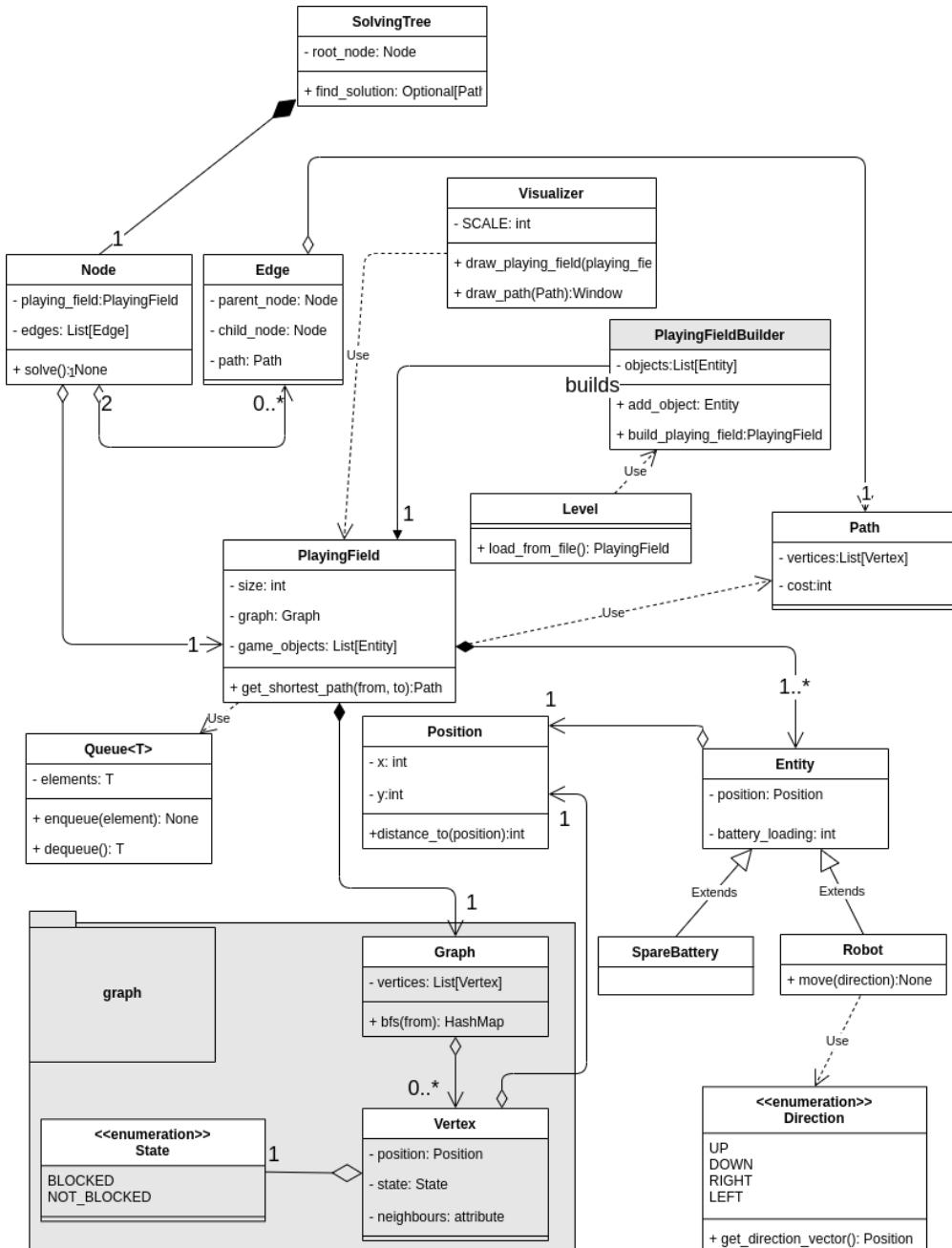


Abbildung 10: UML Klassen Diagramm

3.1.18 Lösungen

Eingabe: Die Dateien enthalten jeweils ein Spielbrett mit den darauf verteilten Batterien und dem Roboter. In der ersten Zeile ist Größe des Spielbretts angegeben, in der zweiten Zeile sind die Koordinaten des Roboters und die Ladung seiner Batterie und in der dritten Zeile ist die Anzahl der restlichen auf dem Spielbrett verteilten Batterien angegeben. Ab der vierten Zeile ist in jeder Zeile eine Batterie angegeben, also ihre Koordinaten und ihre Ladung. Dabei sind die Angaben zu den Koordinaten und der Ladung der Batterien und des Roboters als drei kommagetrennte Werte in der Form "x,y,ladung" geschrieben. Bundesweite-Informatikwettbewerbe [4].

5
3,5,9
3
5,1,3
1,2,2
5,4,3

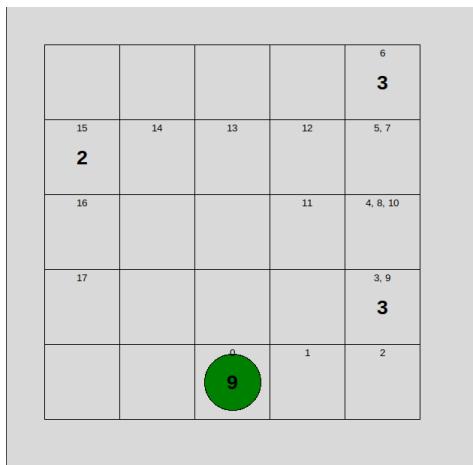


Abbildung 11: Die Lösung des Stromrallye-Spiels

In Abbildung 11 sieht man die Lösung des BWinf Stromrallye Spiels. Die Zahl 9 mit dem grünen Kreis drumherum ist der Roboter, mit der Ladung 9. Die Ersatzbatterien, sind die Fett gedruckten Zahlen in der Mitte der Felder, die Zahl zeigt die Ladung der Ersatzbatterien, beim Start an. Die Ladung des Roboters und der Ersatzbatterien verändert sich während des Spiels, deswegen ist dies bloß eine Momentaufnahme beim Start des Spiels.

Weitere Lösungen befinden sich im Anhang.

3.2 Aufgabe 1 b)

3.2.1 Beschreibung

Bei Teil b) der Aufgabe 1 soll man selbst Stromrallye Spiele erstellen, welche lösbar sind, aber für einen Menschen schwer zu lösen. (Siehe die Aufgabenstellung 5.1 auf Seite18)

3.2.2 Lösungsidee

Es werden zufällig Spielfelder generiert und dann mit dem Programm aus Aufgabe 1 a) geprüft, ob diese lösbar sind. Dabei werden an zufälligen Positionen Ersatz-Batterien mit einer zufälligen Ladung platziert. Die Position und die Ladung, des Roboters wird auch zufällig gewählt. Die Größe des Spielfeld ist ebenfalls zufällig.

3.2.3 Schwierigkeit des Spiels

Das Spiel wird mit mehr Batterien deutlich schwerer zu lösen, auch ein größeres Spiel-feld macht es schwieriger.

3.2.4 Lösung

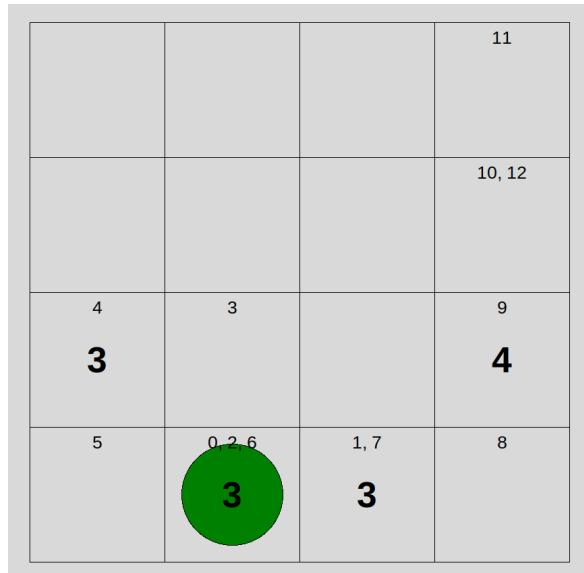


Abbildung 12: Ein generiertes Strom-Rallye Spiel

3.2.5 Umsetzung

Die Informationen zur Umsetzung befinden sich im Anhang.

3.3 Aufgabe 3

3.3.1 Aufgabenstellung

Siehe im Anhang 5.1 auf der Seite 18.

3.3.2 Lösungsidee

Zunächst wandele ich das Straßennetz in einen Graphen um. Dabei werden Straßenkreuzungen zu Knoten, Straßen zu Kanten und die Weglängen zu Kosten der Kanten.

Dann finde ich in diesem Graphen alle möglichen Wege vom Start zum Ziel, allerdings nur Wege ohne Zyklen, also Wege, welche einen Knoten maximal 1 mal besuchen, da alle anderen Wege länger sind. Außerdem würde es sonst

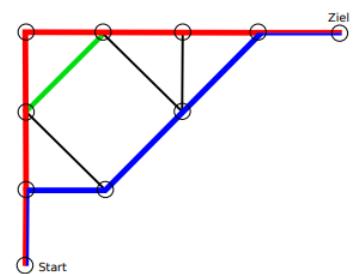


Abb. 13: Straßennetz, rot ist der Weg mit den wenigsten Abkürzungen

unendliche viele mögliche Wege geben, was die Aufgabe unlösbar machen würde.

Es gibt von A nach D, die möglichen Wege $A \rightarrow B \rightarrow D$ und $A \rightarrow C \rightarrow D$.

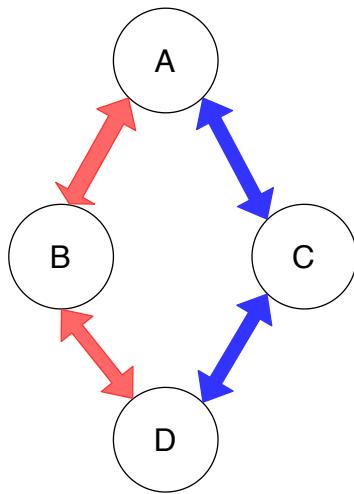


Abbildung 14: Graph

Um alle möglichen Wege zu finden, verwende ich die Tiefe Suche. Wenn man auf einen Knoten stößt, welcher das Ziel ist oder schon besucht wurde, wird abgebrochen.

Problem Dieser Algorithmus braucht sehr lange, weil er alle Wege, die es in einem Graphen gibt, durch geht. An jeder Kreuzung gibt es maximal 8 Straßen, da es Straßen nach oben, unten, rechts, links und diagonale gibt. Dies führt zu einer sehr schlechten worst-case Laufzeit von $O(8^V)$, $V = \text{Anzahl Kanten}$.

Verbesserung Die besten Pfade zu jedem Knoten (Kreuzung) werden in einer HashMap gespeichert. Am Anfang kennen wir nur den Weg zum Startknoten. Nun aktualisieren wir für alle Nachbarn des Startknotens die Map Einträge, dann von diesen Nachbarn wieder und so weiter. Ein Knoten wird nur "betrachtet", wenn ein besserer Weg zu ihm gefunden wurde, also einer, der weniger Abbiegungen hat oder kürzer ist. So werden viel weniger Wege durchsucht.

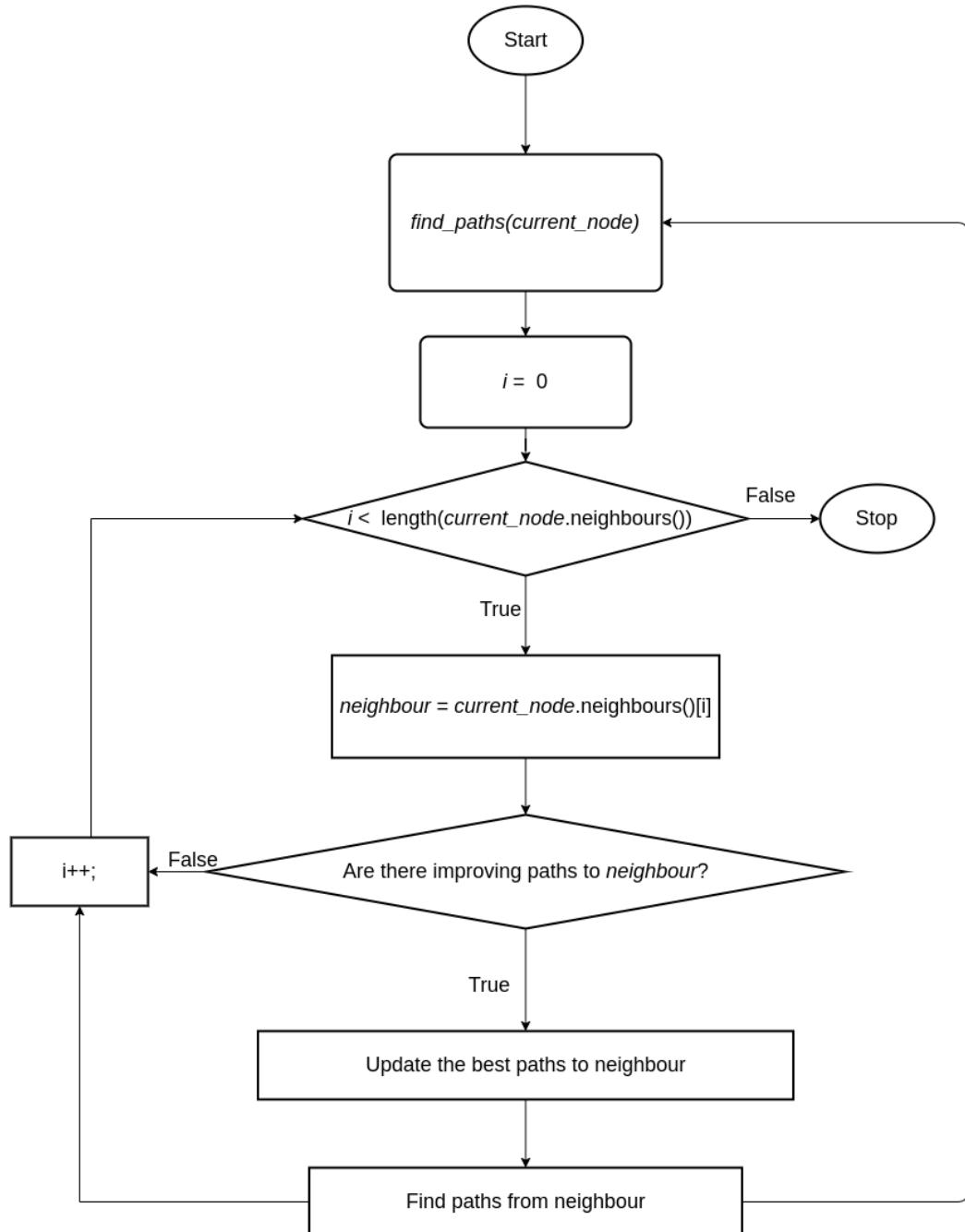


Abbildung 15: Programmablaufplan um den besten Weg zu finden

3.3.3 Abbiegungen bestimmen

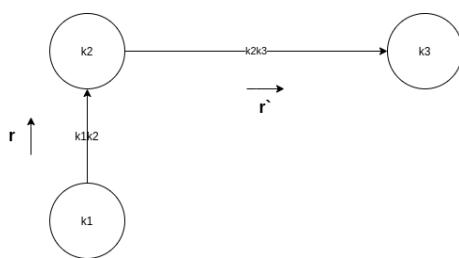


Abbildung 16

Wenn man vom Knoten k_1 zum Knoten k_2 geht und beide eine Position kp_1 und kp_2 im 2-dimensionalen Raum haben, dann bewegt man sich in Richtung $r \rightarrow r = kp_2 - kp_1$ wenn man sich von k_1 zu k_2 bewegt. Ebenso wenn man von k_2 zu k_3 geht, in Richtung $r' \rightarrow r' = kp_3 - kp_2$. Wenn der Einheitsvektor r^0 von r und der Ein-

heitsvektor r'^0 von r' ungleich sind handelt es sich um eine Abbiegung. Wenn die Gleichung $r^0 \neq r'^0$ wahr ist, handelt es sich um eine Abbiegung.

Durch diese Definition lässt sich die Aufgabe auch für ein Straßennetz mit komplizierteren Kreuzungen durchführen.

3.3.4 HashSet

Ein HashSet speichert eine Menge an Daten. Dabei hat es die besondere Eigenschaft, dass es keine Duplikate speichert, jedes Element ist entweder einmal oder kein mal enthalten. Der Vorteil gegenüber einer Liste besteht darin, dass es nur $O(1)$ Zeit benötigt um zu prüfen, ob ein Element enthalten ist, sowie $O(1)$, um ein Element hinzuzufügen.
[11]

3.3.5 HashMap

Eine HashMap speichert key-value Paare, (auch HashTable genannt). Man kann für einen Schlüssel(key) einen entsprechenden Wert finden. Neue Key value Paare können in die HashMap eingetragen werden. Um in einer Hashmap für einen Schlüssel einen Wert zu finden oder ein Wert-Schlüssel-Paar einzufügen, wird nur konstante Zeit benötigt.
[10]

4 Fazit

In die Bearbeitung der beiden Aufgaben habe ich viel Zeit investiert. Obwohl beide Aufgaben an sich sehr unterschiedlich waren, konnte ich für beide Aufgaben ähnliche Algorithmen verwenden. Für beide Aufgaben war die Graphen-Theorie sehr hilfreich. Aufgrund der Seitenbeschränkung in der Facharbeit habe ich mich entschieden, die erste Aufgabe detailliert zu dokumentieren, um die Komplexität und die Vielzahl an Entscheidungen und Überlegungen, die in der Lösung der Aufgabe stecken, zu verdeutlichen. Die 3. Aufgabe konnte ich nicht im gleichen Umfang wie die erste erklären, aber natürlich sind auch hier weitere Lösungen möglich und ähnliche Überlegungen zur Laufzeitoptimierung wie bei der ersten Aufgabe zu berücksichtigen.

5 Anhang

5.1 Offizielle Aufgabenbeschreibung des BWinf

[2]

Aufgabe 1: Stromrallye

Stromrallye ist ein neues Knobelspiel:

Auf einem quadratischen Spielbrett, das in quadratische Felder eingeteilt ist, bewegt sich ein Roboter. Der Roboter hat eine austauschbare Bordbatterie mit vorgegebener Ladung. Auf einigen Feldern liegen außerdem Ersatzbatterien mit unterschiedlichen Ladungen. Die Ladungen aller Batterien sind durch ganze Zahlen gegeben.

Der Roboter kann Schritte um ein Feld nach rechts, links, oben oder unten machen. Bei jedem Schritt sinkt die Ladung seiner Batterie um 1. Sinkt die Ladung auf 0, kann er keine weiteren Schritte machen.

Wenn der Roboter auf ein Feld mit einer Ersatzbatterie kommt, muss er seine Bordbatterie gegen die Ersatzbatterie austauschen, selbst wenn die Ersatzbatterie weniger Ladung hat als die Bordbatterie. Die Ersatzbatterie ist danach die neue Bordbatterie. Nach dem Tausch muss er das Feld verlassen.

Ziel des Spiels Stromrallye ist es, den Roboter so über das Spielbrett zu bewegen, dass am Ende alle Batterien, einschließlich der Bordbatterie, entladen sind.

Hier ist ein Beispiel für eine Spielsituation auf einem Spielbrett mit 5×5 Feldern:



Aufgabe

- Schreibe ein Programm, das Stromrallye spielen kann. Es soll zuerst eine Spielsituation einlesen: die Größe des Spielbretts, die Position des Roboters, die Ladung der Bordbatterie sowie die Positionen und Ladungen der Ersatzbatterien. Danach soll dein Programm bestimmen, ob die eingelesene Spielsituation lösbar ist, also ob der Roboter sich so bewegen kann, dass am Ende alle Batterien entladen sind. Wenn ja, soll dein Programm eine entsprechende Folge von Schritten ausgeben.
- Stromrallye soll auch von Menschen gespielt werden. Schreibe ein Programm, das Spielsituationen erzeugt, die lösbar, aber für einen menschlichen Spieler schwierig zu lösen sind. Dokumentiere, wie du dein Schwierigkeitsmaß definierst.

Hinweis: Zum Ausprobieren kannst du Stromrallye mit einem Damebrett nachspielen. Als Roboter nimmst du einen schwarzen Stein, eine Batterie mit einer Ladung von n ist ein Stapel aus $n + 1$ weißen Steinen. Die Bordbatterie wird auf den Roboterstein gestapelt.

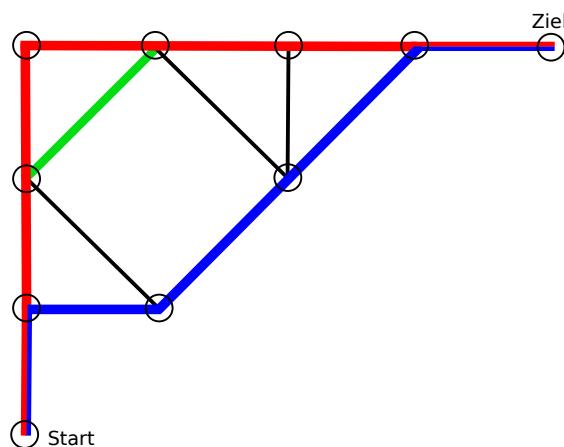
Aufgabe 3: Abbiegen?

Bilal möchte mit seinem Fahrrad zu einem Freund fahren. Der kürzeste Weg führt über viele kleine Straßen und ist verwinkelt. Bilal fährt an Kreuzungen am liebsten geradeaus und ist daher bereit, einen etwas längeren Weg in Kauf zu nehmen, wenn er dafür weniger oft abbiegen muss.

Am unten abgebildeten Beispielwegenetz sieht man, dass weniger Abbiegen tatsächlich zu einer längeren Wegstrecke führen kann. Der kürzeste, blaue Weg hat die Länge $3 + 2 \cdot \sqrt{2} \approx 5,83$ und erfordert dreimaliges Abbiegen. Der rote Weg mit einmaligem Abbiegen und der Länge 7 ist demgegenüber um etwa 20 % länger.

Bei gutem Wetter macht es Bilal nichts aus, einen um 30 % längeren Weg zu fahren. Bei schlechtem Wetter sollte die Verlängerung allerdings maximal 15 % betragen. Bei Sonnenschein ist beim gegebenen Beispiel also der längere rote Weg akzeptabel. Bei Regen ist der rote Weg zu lang, und Bilal ist dann bislang den blauen Weg gefahren.

Neulich hat Bilal aber entdeckt, dass er den roten Weg abkürzen kann, indem er die grüne Straße nimmt. Dieser rot-grüne Weg ist $5 + \sqrt{2} \approx 6,41$ lang und damit um etwa 10 % länger als der kürzeste Weg. Bei schlechtem Wetter fährt Bilal also nun den rot-grünen Weg, auf dem er einmal weniger abbiegen muss als auf dem blauen Weg.



Aufgabe

Hilf Bilal und schreibe ein Programm, das eine Straßenkarte mit Start- und Zielpunkt einliest und ihm einen Weg vorschlägt. Damit das Programm bei jedem Wetter nutzbar ist, soll der Benutzer die maximale Verlängerung in Prozent eingeben können.

5.2 Programm Erklärung Aufgabe 1

5.2.1 SolutionTree

Beschreibung: Der *SolutionTree* löst ein Spielfeld, er speichert die *root_node* (siehe 5.2.2 auf Seite 21.) Diese *root_node* hat wiederum Kind-Knoten, welche auch wieder Kind-Knoten haben usw. . Mit der Funktion *find_solution* wird eine Lösung für das entsprechende Spielfeld (das Spielfeld wird nicht im *SolutionTree* selbst gespeichert, aber in der *root_node*) gefunden, falls eine existiert.

Beziehungen zu anderen Klassen:

- hat Objekte von *Node* (5.2.2 siehe auf Seite 21)

wichtige Attribute:

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>root_node</i>	Node	Die <i>root_node</i> ist der oberste Knoten im <i>SolutionTree</i> , sie hat selbst wieder Kind-Knoten welche wieder Kind-Knoten haben usw.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>find_solution()</i>	Optional[Path]	Die <i>find_solution</i> Methode findet einen Pfad, welcher das Spielfeld, der <i>root_node</i> löst, falls es lösbar ist.

5.2.2 Node

Beschreibung: Die *Node*-Klasse ist Teil des *Solution Tree* (siehe 5.2.1 auf Seite 21) . Außerdem speichert sie einen Spielstand des Stromrallye. Die *Node* hat Kanten zu allen möglichen Knoten mit Spielständen, welche durch das Bewegen vom Roboter zu einer Ersatzbatterie möglich sind. **Beziehungen zu anderen Klassen:**

- ist Teil vom *SolutionTree* (siehe 5.2.1 auf Seite 21)
- hat Objekte von *Edge* (5.2.3 siehe auf Seite 22)
- hat Objekte von *PlayingField* (?? siehe auf Seite ??)

wichtige Attribute:

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>edges</i>	List[Edge]	Die <i>edges</i> sind die Kanten zu den Kind-Knoten der <i>Node</i> (siehe 5.2.3 auf Seite 22)
. -	<i>playing_field</i>	PlayingField	Das <i>playing_field</i> ist der Spielstand dieses Knotens.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>solve()</i>	None	Die <i>solve()</i> Funktion, löst den Spielstand des Knotens (<i>playing_field</i>). Dabei ruft die <i>solve</i> Funktion rekursiv, wieder <i>solve</i> für alle Kind-Knoten des Knotens auf (siehe Pseudocode).

5.2.3 Edge

Beschreibung: Die *Edge* Klasse ist die Kante welche Knoten im *SolutionTree* (siehe 5.2.1 auf Seite 21) miteinander verbindet. Es wird der Pfad gespeichert, welcher der Roboter laufen muss um den Zustand des Spielfelds von Eltern- zu Kind-Knoten zu verändern. **Beziehungen zu anderen Klassen:**

- ist Teil des *SolutionTrees*
- hat Objekte von *Node* (5.2.2 siehe auf Seite 21)
- hat Objekte von *Path* (?? siehe auf Seite ??)

wichtige Attribute:

wichtige Methoden:

KEINE wichtigen Methode

5.2.4 Entity

Beschreibung: Die *Entity* Klasse, beinhaltet alle Dinge, welche Roboter und Ersatzbatterie gemeinsam haben. Roboter und Ersatzbatterie erben beide von Entity. Beide haben eine Position und einen Ladestand. **wichtige Attribute:**

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>path</i>	Path	Die <i>path</i> -Variable, gibt an, welcher Pfad der Roboter gehen muss um den Zustand des Spielfelds vom Eltern-Knotens-Spielstand zum Kind-Knotens-Spielstand zu verändern.
-	<i>parent_node</i>	Node	Die <i>parent_node</i> speichert den Eltern-Knoten
-	<i>child_node</i>	Node	Die <i>child_node</i> speichert den Kind-Knoten

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>position</i>	Position	Die <i>position</i> gibt die Position der Entity auf dem Spielfeld an
-	<i>battery_loading</i>	int	Der aktuelle Batterie-Ladestand der Batterie

wichtige Methoden:

KEINE wichtigen Methode

5.2.5 Position

Beschreibung: Die **Position** Klasse, speichert die Position der entitys (Roboter, Ersatzbatterien). Da wir uns im 2-dimensionalen Raum befinden, besteht die Position aus x und y Koordinate, welche Attribute der Position Klasse sind.

Dabei habe ich ein Kordinatensystem gewählt, indem oben links ist. Nach unten steigt y, nach rechts steigt x. Die Positions Klasse könnte man auch als Vektor bezeichnen. Sie implementiert die gängigsten Vektor-Funktionen, darunter (plus, minus, Länge des Vektors, Multiplikation ...)

wichtige Attribute:

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>x</i>	int	Die x-Koordinate der Position.
-	<i>y</i>	int	Die y-Koordinate der Position.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>manhattan_distance(other_position)</i>	int	manhattan Distanz zur anderen Position. Die Manhattan Distanz wird folgendermaßen berechnet, $ x_1 - x_2 + y_1 - y_2 $.

5.2.6 Roboter

Beschreibung: Der Roboter erbt von der Entity Klasse. Zusätzlich hat der Roboter aber noch Methoden um sich zu bewegen und um seine Batterie mit einer Ersatzbatterie auszutauschen.

wichtige Attribute:

erbt die Attribute *position* und *battery_loading* von *Entity*

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>move(direction)</i>	None	Der Roboter kann sich bewegen.
-	<i>change_battery()</i>	None	Der Roboter wechselt die Batterie, mit der unter sich liegenden Ersatzbatterie. Die Methode ist <i>privat</i> und wird vom Roboter aufgerufen, wenn er sich auf ein Feld mit einer Ersatzbatterie bewegt.

5.2.7 Enum-Directions

Beschreibung: Das Enum Description, gibt die vier Richtungen (oben, unten, rechts, links) an, in welche sich der Roboter auf dem Stromrallye-Spielfeld bewegen kann. Die 4 verschiedenen Werte, welcher das Enum Directions haben kann sind: **Directions.UP**

Directions.DOWN

Directions.LEFT

Directions.RIGHT

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<code>get_direction_vector()</code>	Position	Die Methode getdirectionvector gibt, den entsprechenden Richtungsvektor der Richtung zurück- UP => (0, -1)- DOWN => (0, 1)- RIGHT => (1, 0) - LEFT => (-1, 0). In diesem Fall ist unsere Positions-Klasse, der Richtungsvektor

5.2.8 SpareBatterie

Beschreibung: Die Ersatzbatterie (SpareBattery) erbt von der Entity-Klasse. Da sie sich nicht bewegen kann und auch sonst nichts machen kann, hat diese keine besonderen Attribute oder Funktionen.

wichtige Attribute:

erbt `position` und `battery_loading` von `Entity` wichtige Methoden:

KEINE wichtigen Methode

5.2.9 Spielfeld

Beschreibung: Die Spielfeld Klasse beinhaltet alle Objekte des Spiels. Sie speichert die Elemente des Spiels. Außerdem auch das Spielfeld als Graphen. Sie hat eine Größe, da sie quadratisch ist, wird nicht zwischen Länge und Breite unterschieden.

wichtige Attribute:

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>size</i>	int	Die Größe des Spielfelds
-	<i>game_objects</i>	List[Entity]	Die Liste aller Objekte, die auf dem Spiel-feld sind.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>get_shortest_path(-from:Position, to:Position)</i>	Path	Findet den kürzesten Weg auf dem Spiel-feld von einer Position zu einer anderen, ohne andere Ersatzbatterien aufzusammeln außer des Ziels.

5.2.10 Graph

Beschreibung: Die Graph Klasse, entspricht einem normalen Graphen in der Informatik. Es ist ein ungerichteteter, ungewichteter Graph, weswegen es auch keine Kanten als Objekte gibt, auch keine Klasse, da alle Kanten Kosten 0 haben. Stattdessen speichert jeder Knoten(Vertex), die Nachbar Knoten(Vertices). Der Graph speichert alle Knoten in einer Liste.

wichtige Attribute:

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>vertices</i>	HashMap-[Position,-Vertex]	Alle Knoten des Graphen werden in der HashMap <i>vertices</i> gespeichert. Es wird eine HashMap verwendet, da man bei solcher nur konstante Zeit O(1) benötigt um ein Knoten an einem Punkt zu bekommen. Eine weitere Alternative ist ein 2-dimensionales Array, bei dem an Stelle <i>vertices[x][y]</i> -> der Knoten der Position x y ist.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>bfs(from, to)</i>	HashMap[Vertex] Path	Führt die breiten Suche (engl. breadth first search) aus.

5.2.11 Vertex

Beschreibung: Die Vertex-Klasse, ist ein Knoten des Graphen. Der Knoten kennt seine Nachbar Knoten und hat eine Position. **wichtige Attribute:**

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>position</i>	Position	Position auf dem Spielfeld.
-	<i>neighbours</i>	List[Vertex]	Nachbarknoten

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>add_neighbour</i>	None	Fügt einen Knoten als Nachbar hinzu.

5.3 Umsetzung Aufgabe 1 b)

Die Implementierung ist wie in Aufgabe 1 Teil a) in Python. Aus dem Grund, dass ich den Algorithmus aus Teil 1 nutze, habe ich nur eine Klasse benötigt und zwar **PlayingFieldGenerator**

PlayingFieldGenerator

Beschreibung: Die PlayingFieldGenerator- Klasse ist die einzige Klasse, des 2.Teils. Diese erzeugt ein zufälliges, lösbares Stromrallye Spiel **wichtige Attribute:**

Sichtbarkeit	Attribute	Typ	Erklärung
-	<i>playing_field_builder</i>	Playing- Field- Builder	baut das Spielfeld
-	<i>positions_used</i>	HashSet- [Position]	Speichert alle Positionen an welchen bereits ein Objekt ist, damit nicht zwei Objekte an einem Ort sind. Das HashSet eignet sich dafür, da es nur $O(1)$ Zeit benötigt.

wichtige Methoden:

Sichtbarkeit	Methoden	Rückgabe-Typ	Erklärung
+	<i>create_solveable_playing_field</i>	PlayingField	Die Methode erzeugt ein lösbares Spielfeld

5.4 Beispiel3

5.4.1 Eingabe

14
3,5,9
3
6,4,4
5,12,10
6,2,5

5.4.2 Lösung

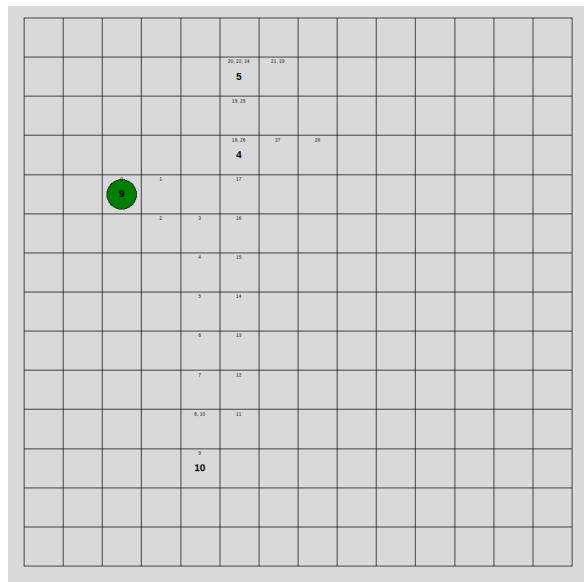


Abbildung 17: Lösung des BWinfs Stromrallye Spiels Beispiel 3

5.5 Beispiel4

5.5.1 Eingabe

100
40,25,20
0

5.5.2 Lösung

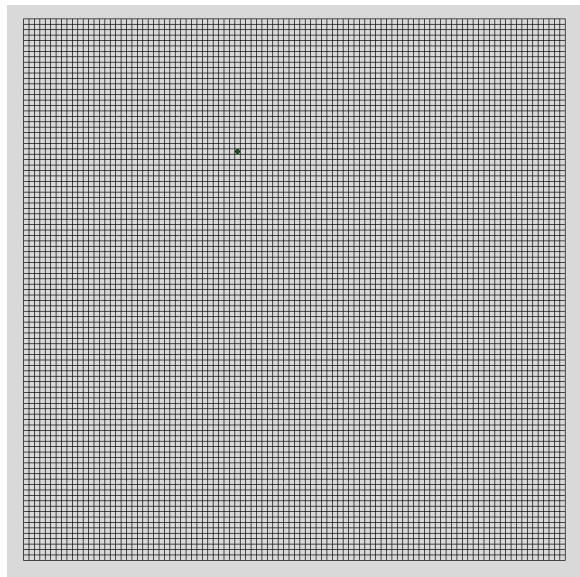


Abbildung 18: Lösung des BWinfs Stromrallye Spiels Beispiel 4

Da auf Grund der Größe wenig zu erkennen ist. Hier heran gezoomt.

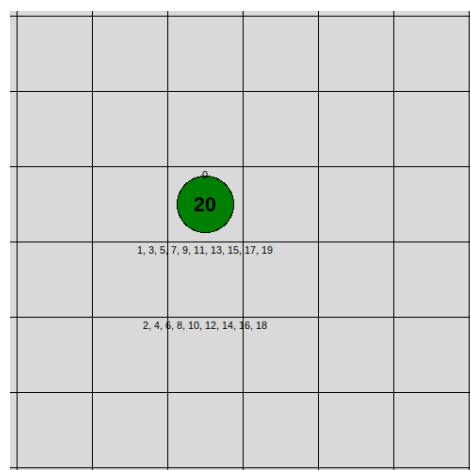


Abbildung 19: Lösung des BWinfs Stromrallye Spiels Beispiel 4 heran gezoomt

6 Quellen

Literatur

Digital

- [1] Bundesweite-Informatikwettbewerbe. *Bundeswettbewerb Informatik: Talente entdecken, Talente fördern.* 2019. URL: <https://bwinfo.de/bundeswettbewerb/>. (accesed: 14.8.2020).
- [2] Bundesweite-Informatikwettbewerbe. *Die Aufgaben der 2. Runde.* 2019. URL: <https://bwinfo.de/fileadmin/bundeswettbewerb/38/aufgaben382.pdf>. (accessed: 06.07.2020).
- [3] Bundesweite-Informatikwettbewerbe. *Plakat 38. Bundeswettbewerb Informatik.* 2019. URL: https://bwinfo.de/fileadmin/bundeswettbewerb/38/Plakat_BwInf_38.pdf. (accessed: 06.07.2020).
- [4] Bundesweite-Informatikwettbewerbe. *BWINF: Material 38.2.* 2020. URL: <https://bwinfo.de/bundeswettbewerb/38/2/material/>. (accessed: 04.08.2020).
- [5] wikipedia. *Multigraph: Mehrfachkanten werden durch eine gewichtete Kante visualisiert.* 2020. URL: https://de.wikipedia.org/wiki/Graph_%C2%8Graphentheorie%C2%9#/media/Datei:CPT-Graphs-directed-weighted-ex1.svg. (accessed: 04.08.2020).
- [6] wikipedia. *Plan der Wiener U-Bahn.* 2020. URL: https://de.wikipedia.org/wiki/Graph_%C2%8Graphentheorie%C2%9#/media/Datei:U-Bahn_Wien.png. (accessed: 04.08.2020).

Bücher

- [7] Edsger W Dijkstra u. a. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), S. 269–271.
- [8] Peter E Hart, Nils J Nilsson und Bertram Raphael. “Correction to ’A formal basis for the heuristic determination of minimum cost paths’”. In: *ACM SIGART Bulletin* 37 (1972), S. 28–29.
- [9] Thomas H Cormen u. a. “Introduction to algorithms”. In: MIT press, 2009, S. 594–603.
- [10] Thomas H Cormen u. a. “Introduction to algorithms”. In: MIT press, 2009, S. 253–280.
- [11] David Drohan. “Data Structures”. In: (2012), S. 57–60.

- [12] Béla Bollobás. *Modern graph theory*. Bd. 184. Springer Science & Business Media, 2013.

7 Erklärung über die selbständige Anfertigung der Arbeit

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, alle aus anderen Werken wörtlich oder sinngemäß entnommenen Stellen und Abbildungen unter Angabe der Quelle als Entlehnung kenntlich gemacht und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Mainz, 15. August 2020 Tobias Bück