

Aufgabe 1: Stromrallye

Teilnahme-Id: 52586

Bearbeiter dieser Aufgabe:
Michal Boron

April 2020

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Definitionen	1
1.2	Graph	2
1.2.1	BFS	2
1.2.2	Schleifen in Knoten	3
1.3	Backtracking	4
1.3.1	Klassifizierung des Problems	4
1.3.2	Erreichbarkeit	5
1.3.3	Rekursion	5
1.4	Laufzeit	6
1.5	Generierung von Spielsituationen	8
1.5.1	Laufzeit	8
1.6	Erweiterungen	8
2	Umsetzung	9
2.1	Die Klasse <code>Graph</code>	9
2.2	Die Klasse <code>Backtracking</code>	10
2.3	Die Klasse <code>Generator</code>	12
3	Beispiele	13
3.1	Beispiel 0 (BWINF)	13
3.2	Beispiel 1 (BWINF)	13
3.3	Beispiel 2 (BWINF)	13
3.4	Beispiel 3 (BWINF)	14
3.5	Beispiel 4 (BWINF)	14
3.6	Beispiel 5 (BWINF)	14
3.7	Beispiel 6	14
3.8	Beispiel 7	14
3.9	Beispiel 8	15
3.10	Beispiel 9	15
4	Quellcode	16

1 Lösungsidee

1.1 Definitionen

Als eine *Batterie* bezeichne ich ein Objekt mit zwei Koordinaten x, y und einer Ladung c . Unter einer *Ladung* versteht man eine nichtnegative, rationale Zahl. Bei Koordinaten sowie bei einer Ladung konzentrieren wir uns auf nichtnegative, ganze Zahlen.

Gegeben seien eine zweidimensionale, quadratische Matrix M mit der Seitenlänge l , eine Menge von Batterien B und eine Startbatterie s .

Jeweilige Batterie b_i aus der Menge B besitzt zwei Koordinaten $x_i \leq l$ und $y_i \leq l$ und eine Ladung c_i .

In der Matrix gibt es *Felder*. Jedes Feld ist eine Kombination aus einer x - und einer y -Koordinate. Das bedeutet, dass jede Batterie auch auf einem Feld liegt.

Nach der Aufgabenstellung dürfen wir einen *Schritt* zwischen zwei Feldern machen. Dieser Schritt ist ein Übergang von einem Feld zu einem anderen. Nach der Aufgabenstellung dürfen wir Schritte nach links, rechts oben und unten machen. Angenommen, stehen wir auf einem Feld f mit Koordinaten (x_f, y_f) . Wir dürfen einen Schritt

- nach links zum Feld mit einer x -Koordinate um 1 kleiner als x_f , also zum Feld (x_{f-1}, y_f) ,
- nach rechts zum Feld mit einer x -Koordinate um 1 größer als x_f , also zum Feld (x_{f+1}, y_f) ,
- nach oben zum Feld mit einer y -Koordinate um 1 kleiner als y_f , also zum Feld (x_f, y_{f-1}) ,
- nach unten zum Feld mit einer y -Koordinate um 1 größer als y_f , also zum Feld (x_f, y_{f+1}) ,

machen.

Wir können nun feststellen, dass

Beobachtung 1 *die minimale Anzahl der Schritte von einem Feld, auf dem eine Batterie i liegt, zu einem Feld, auf dem eine Batterie j liegt, konstant ist.*

Die minimale Anzahl der Schritte, die man von einem Feld p zu einem Feld q machen muss, nenne ich die *Entfernung zwischen p und q* oder *Entfernung von p zu q* .

Die Anzahl der Schritte, die wir machen dürfen, ist durch die Größe der Ladung determiniert. Wir starten auf dem Feld der Startbatterie, auch *Startfeld* genannt. Laut der Aufgabenstellung nehmen wir die Ladung der Batterie, auf Feld deren, wir momentan stehen und die Größe dieser Ladung der Anzahl der Schritte entspricht, die wir momentan machen dürfen. Eine solche Ladung bezeichne ich als die *aktuelle Ladung*. Diese Ladung verkleinert sich um 1 mit jedem gemachten Schritt.

Wenn die Anzahl der Schritte reicht, um ein anderes Feld mit einer Batterie zu erreichen, müssen wir unsere aktuelle Ladung a sofort gegen die auf dem Feld liegenden Ladung b austauschen. Dann lassen wir die Ladung a auf dem Feld der Ladung b liegen. b wird zu unserer nächsten aktuellen Ladung.

Jede Batterie besitzt auch einen *Eingabeindex*, der beim Einlesen zugeordnet ist. Diese Indizes entsprechen der Reihenfolge der Batterien in der Eingabe. Die Startbatterie besitzt stets den Eingabeindex 0.

1.2 Graph

Wir ordnen jedem Feld in der Matrix einen *Brettindex* zu. Wir legen eine neue Menge fest: V . In dieser Menge befinden sich alle Brettindizes der Felder in M . Außerdem legen wir eine andere Menge fest: E . Wir iterieren durch alle Felder in M . Für jedes Feld i werden die Brettindizes seiner benachbarten Felder bestimmt, also derjenigen, zu denen man von i einen Schritt machen kann. In der Menge E wird jeweils die Verbindung zwischen i und einem Nachbarfeld mit Hilfe der Indizes von i und dem Nachbarn gespeichert. Jedes Feld kann dementsprechend maximal 4 Nachbarfelder besitzen.

Anhand der festgelegten Mengen lässt sich ein ungerichteter, ungewichteter Graph $G(V, E)$ bilden. In diesem Graphen ist jeder Knoten ein Feld aus der Matrix M . Jede Kante ist demzufolge ein Schritt zwischen zwei Feldern in M . Sie hat stets das Gewicht 1.

1.2.1 BFS

Anhand der Beobachtung 1 lässt sich feststellen, dass diese Entfernungen zwischen Feldern, auf denen Batterien liegen, sich durch einen Lauf vom Breitensuche-Algorithmus (engl. *breadth-first search*, *BFS*) einfach bestimmen lassen.

Für eine Batterie b können wir auf folgender Weise die Entfernungen zu allen anderen Batterien bestimmen. Sie werden in einer zweidimensionalen Tabelle A gespeichert. Bei jeder iterierten Stelle $A_{i,j}$ in

der Tabelle A entsprechen i der Index der Batterie, von der die Entfernung gemessen wird, und j der Index der anderen Batterie, deren Entfernung von i bestimmt wird. Dementsprechend ist $A_{i,j}$ dann die minimale Entfernung von i zu j .

Es gibt auch eine parallele, zweidimensionale Tabelle A' , deren Funktion ich im nächsten Abschnitt erläutere.

Wir finden im Graphen G den Knoten, der dem Feld, auf dem b liegt, entspricht. Wir fügen diesen Knoten mit einer Entfernung 0 in eine Warteschlange ein. Diese Warteschlange wird iteriert, so lange es noch Knoten gibt. Jeden iterierten Knoten nennen wir i und seine entsprechende Entfernung von b : d_i .

Wie nehmen i aus der Warteschlange heraus. Danach wird durch die Nachbarknoten von i iteriert, dass heißt, durch diejenigen, die eine Kante mit i besitzen. Jeden iterierten, benachbarten Knoten nenne ich an dieser Stelle j . Es wird überprüft, ob j bereits besucht wurde. Wenn ja, wird der Knoten $j+1$ genommen. Wenn nicht, dann wird überprüft, ob der Knoten j einem Feld entspricht, auf dem eine Batterie liegt. Wenn nicht, wird j ganz normal in die Warteschlange mit der Entfernung $d_i + 1$ eingefügt. j wird auch danach als besucht markiert.

Wenn aber der Knoten j einem Feld entspricht, auf dem eine Batterie liegt, wird zuerst geprüft, ob $A_{i,j}$ bereits gleich 1 oder 2 ist (mehr dazu im nächsten Abschnitt). Wenn nicht, wird $A_{i,j}$ der Wert $d_i + 1$ zugewiesen. Nun, nur wenn $d_i + 1 > 2$ wird der Knoten j als besucht gekennzeichnet.

Wenn aber $A_{i,j}$ bereits gleich 1 oder 2 ist, wird geprüft, ob $A'_{i,j}$ bereits einen Wert besitzt. Wenn ja und wenn $d_i + 1 < A'_{i,j}$, wird der Wert $A'_{i,j}$ als $d_i + 1$ aktualisiert. Wenn es früher keinen Wert $A'_{i,j}$ nicht gab, wird er als $d_i + 1$ gespeichert.

Wir sollen bemerken, dass wir

Beobachtung 2 auf dem Weg von einer Batterie p zur Batterie q nicht immer die minimale Anzahl an Schritten $A_{p,q}$ machen können, wenn die Größe der Ladung genügend ist, um das zu tun.

Es ist auch zu bemerken, dass

Beobachtung 3 wenn die minimale Anzahl der Schritte $A_{p,q} > 2$ oder $A'_{p,q} > 2$ ist, kann man stets die Batterie q auch in $A_{p,q} + 2k$, bzw. in $A'_{p,q} + 2k$, wobei $k \in \mathbb{N}^+$, Schritten erreichen, so lange die Größe der aktuellen Ladung das erlaubt.

Der Beweis der Beobachtung 3 ist trivial: wenn wir auf einem Feld f stehen, können nach demselben Feld f mit genau 2 Schritten zurückkehren, wenn wir zu einem benachbarten Feld h einen Schritt machen und dann von h zu f zurück.

So kann man auch die Beobachtung 2 beweisen, indem man bemerkt, dass wir auf dem Weg von p zu q einfach einen Schritt zurück und nach vorne machen.

Genau aus dem Grund musste ich auch zusätzlich mit Hilfe der Tabelle A' prüfen, ob es auch bei den Entfernungen nicht größer als 2 einen anderen Weg gibt, der mindestens der Länge 3 ist. Tabelle A' speichert die Länge der minimalen Entfernung von einer Batterie zu einer anderen, die auch länger ist als 2. Abbildung 1.2.1 präsentiert ein Beispiel, in dem diese besondere Unterscheidung wichtig ist.

5		5	2
1		1	

Abbildung 1: Dargestellt sind Fragmente einer Matrix. Die Zahlen stellen die Ladungen der Batterien dar, die auf diesen Feldern liegen. Im ersten Beispiel kann man vom Feld mit 5 das Feld mit 1 in minimaler Anzahl von einem Schritt erreichen. Man kann auch aber dieses Feld in 3 Schritten erreichen, wenn man einen Schritt nach rechts, dann nach unten und nach links macht. Außerdem kann man laut der Beobachtung 3 auch dieses Feld in 5 Schritten erreichen. Im zweiten Beispiel kann man nur das Feld mit 1 in einem Schritt erreichen, weil es keinen andren Weg vom Feld mit 5 zum Feld mit 1 gibt, der länger ist als 2.

1.2.2 Schleifen in Knoten

Wir müssen auch einen Sonderfall berücksichtigen, dass man von einem Knoten zu demselben Knoten zurückkehren kann. Mit Hilfe der Tabellen A und A' können wir es nicht überprüfen, ob es überhaupt

möglich ist.

Um eine Schleife in Knoten b durchzuführen, brauchen wir mindestens ein benachbartes Feld i , auf dem keine Batterie liegt. Mit Hilfe dieses Feldes können wir einen Schritt von b zu i machen und von i zu b zurück. Die Anzahl der Schritte ist $l = 2$.

Wenn wir aber sicherstellen, dass es neben einem batteriefreien Feld i ein anderes batteriefreies Feld j gibt ($i \neq j$), können wir eine Schleife in b erstellen, deren Länge $l = 4$ ist. Auf folgender Weise machen wir die Schritte: $b \rightarrow i \rightarrow j \rightarrow i \rightarrow b$. Auch hier können wir die in der Beobachtung 3 bemerkte Ordnungsgemäßheit anwenden, wenn wir l statt $A_{i,j}$ nehmen. (s. Abb. 1.2.2). Wir legen noch ein Array T fest, in dem wir die Information an der Stelle T_i speichern, ob die Batterie i zwei, eins oder null zusätzliche, freie Nachbarfelder hat.

1	4	1
1		1
2	5	1

1	4	1
1		1
1		1
2	5	1

1	4	1
1	2	1
2	5	1

Abbildung 2: Dargestellt sind Fragmente einer Matrix. Die Zahlen stellen die Ladungen der Batterien dar, die auf diesen Feldern liegen. Im ersten Beispiel kann man maximal zwei Schritte von der Batterie mit der Ladung 4 machen, um an dieselbe Stelle zurückzukommen. Im zweiten Beispiel kann man schon $2n$ Schritte machen, wobei $n \in \mathbb{N}^+$. Im dritten Beispiel kann man überhaupt keine Schleife durchführen.

Auf diese Weise bekommen wir eine Tabelle A mit allen minimalen Entfernungen zu Batterien, die von jeder Batterie erreichbar sind. In der Tabelle A' haben wir die zusätzlichen minimalen Entfernungen, wenn der entsprechende Wert in A kleiner ist als 3. Außerdem haben wir ein Array T , in dem die Anzahl der zusätzlichen, batteriefreien Felder jeder Batterie gespeichert ist.

1.3 Backtracking

Unsere Aufgabe bleibt nun, so einen Pfad von der Startbatterie s zu finden, der durch alle Batterien führt und am Ende alle Ladungen 0 betragen. Außerdem muss man unbedingt den Aspekt beachten, dass man von einer Batterie zu einer anderen übergehen darf, nur wenn die aktuelle Ladung nicht kleiner ist als die Entfernung zwischen den beiden. Darüber hinaus wird die übriggebliebene Ladung mit der an der Stelle der Zielbatterie liegenden Ladung ausgetauscht. Wir können auch sofort bemerken, dass es sich gar nicht lohnt, von einer Batterie zu einem Feld, auf dem keine Batterie liegt, zu übergehen, weil wir einfach nicht fortsetzen könnten. Der Sonderfall ist hier natürlich die Situation am Ende der Aufgabe, wenn alle Batterien besucht sind und alle liegenden Ladungen 0 betragen, aber wir mit der aktuellen Ladung übriggeblieben sind. Dann müssen wir die aktuelle Ladung noch verbrauchen und es ergibt genauso viel Sinn zu einem batteriefreien sowie zu einem Feld mit einer Batterie zu übergehen.

Wenn wir diese Situation ausschließen, können wir bemerken, dass wir stets von einer Batterie zu einer anderen übergehen müssen.

Wir können auch bemerken, dass dieses Problem sehr komplex ist. Schauen wir auf ein vereinfachtes Beispiel dieser Aufgabe.

1.3.1 Klassifizierung des Problems

Angenommen, sind alle Werte in A 1 und alle Batterien haben auch Ladungen 1 (wie im Beispiel 3.2). Nun ist das Ziel der Aufgabe, einen Hamiltonpfad in diesem entstandenen Graphen zu finden. O.b.d.A. können wir annehmen, dass es sich um einen Hamiltonpfad und keinen Hamiltonzyklus handelt. Wir starten üblicherweise von der Batterie mit Index 0 und gehen zu nächsten Batterien. Wir müssen jedoch anmerken, dass wir bei der letzten Batterie die Ladung zur nächsten Batterie nicht weiterleiten können, weil das schon das Ende der Matrix ist. In diesem Falle können wir einfach die von der letzten Batterie zur vorletzten gehen. Wir zählen dann auch nicht das vorletzte Feld als zweimal besucht.

Das Hamiltonkreisproblem ist ein NP-vollständiges Problem. Wenn wir mehrere verschiedenen Ladungen und mehr Verbindungen unterschiedlicher Länge hinzufügen, wird das Problem komplexer. Daraus

können wir schlussfolgern, dass es sich in unserem Problem auch um NP-Vollständigkeit handelt.

Wir sollen auch bemerken, dass die Größe der Eingabe in den Beispielen sehr klein bleibt. Aus diesem entschied ich mich für eine weniger effiziente, aber sehr genaue Methode, um diese Aufgabe zu lösen: Backtracking.

Wir betrachten ein Array C , die die Ladungen jeder Batterie enthält. An jeder Stelle i in diesem Array wird die aktuelle Ladung einer Batterie mit dem Eingabeindex i gespeichert.

1.3.2 Erreichbarkeit

Wir sollen zuerst definieren, was wir unter *Erreichen* einer Batterie c von einer b aus verstehen. Eine Batterie c kann von einer Batterie b aus erreicht werden, wenn die aktuelle Ladung nicht kleiner ist als die Entfernung an der Stelle $A_{b,c}$. Das bedeutet einfach, dass die Ladung ausreicht, um die Schritte von b zu c zu machen. Außerdem soll c in dem Array C eine Ladung größer als 0 besitzen. So ist c *erreichbar*. Ein Sonderfall in dieser Definition ist natürlich eine Schleife von b zu sich selbst. In diesem Fall muss der Wert an der Stelle b in der Matrix T mindestens 1 betragen und die aktuelle Ladung muss mindestens 2 sein.

1.3.3 Rekursion

Ich bildete eine rekursive Funktion, die alle möglichen Verbindungen zwischen jeweiligem Paar von Batterien untersucht. Bei jedem Lauf der Funktion wird untersucht, zu welcher Batterie und mit wie vielen Schritten übergangen wird.

Wenn es möglich ist, von einer Batterie a aus eine Batterie b in c Schritten zu erreichen, lässt man diese rekursive Funktion dann an der Batterie b laufen. Wenn es nicht geht, an einer Stelle fortzusetzen, wird abgebrochen und man geht zurück zu der Batterie, an der noch eine Möglichkeit entsteht, von dieser Batterie einen anderen Weg zu nehmen und mit einer anderen Anzahl von Schritten und/oder zu einer anderen Batterie zu übergehen. Wenn es zu einem Zustand kommt, dass alle Batterien entladen sind, also alle Stellen in C 0 betragen, wird dieser Stand als ein Ergebnis behandelt und ausgegeben.

Wir können die Funktionsweise dieser Funktion in Analogie zur Tiefensuche (engl. *depth-first search*, *DFS*) stellen. In diesem Falle haben wir mit einem großen Baum mit einer Wurzel in s zu tun. Jede Kante in einem solchen Baum entspricht einer möglichen Verbindung von einer Batterie zu einer anderen mit einer bestimmten Anzahl von Schritten. Diese Anzahl an Schritten kann dementsprechend als ein Gewicht an einer Kanten gelten. Die Blätter in einem solchen Baum sind natürlich Batterien, die von einer anderen Batterien an einem niedrigeren Niveau liegt.

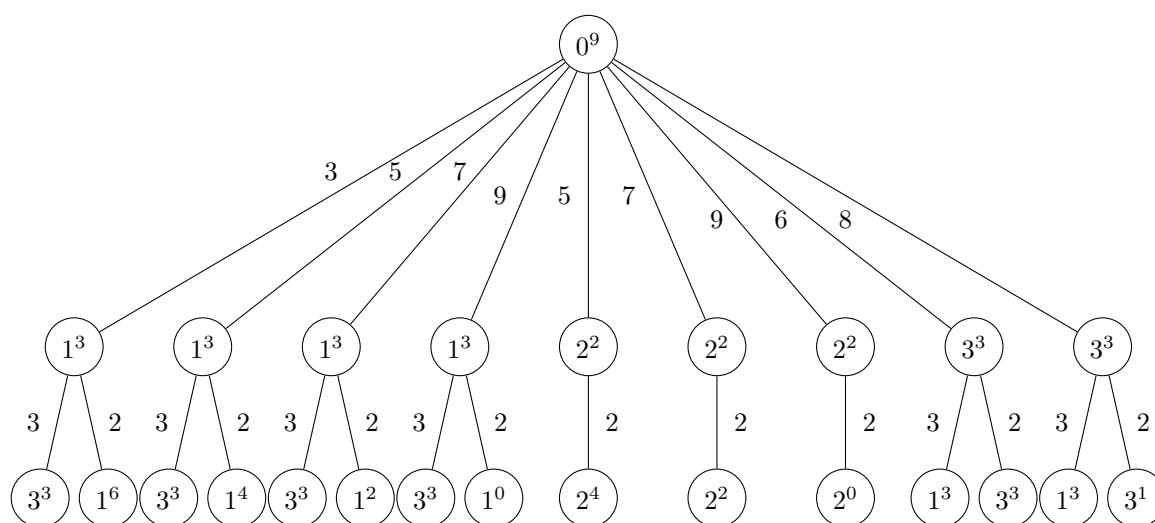


Abbildung 3: Ein abgebildeter Baum der 0., 1. und 2. Niveaus des Beispiels 3.1 (das Beispiel aus der Aufgabenstellung). Die Nummern in Knoten entsprechen den Eingabeindizes der Batterien. Die Potenzen stellen die Ladung dar, die wir auf einem Feld finden, wenn wir eine Batterie erreichen.

Diese rekursive Funktion nimmt als Parameter einen Eingabeindex id , die aktuelle Ladung, ein kopiertes Array C und ein Array R , das die Information über einen aktuellen Pfad von der Batterie 0 zur Batterie id speichert. Das Array R speichert die Eingabeindizes der Batterien, die bereits besucht wurden, mit der entsprechenden Anzahl von Schritten, die gemacht wurden, um diese Zielbatterie zu erreichen.

Am Anfang der Funktion werden alle erreichbaren Batterien bestimmt. Ich forme dadurch ein Array N_{id} , das alle erreichbaren Nachbarn von id , also die Eingabeindizes der Batterien, mit ihren entsprechenden minimalen Entfernungen enthält. Bei der Bildung dieses Arrays stelle ich auch sicher, dass diese Batterien nach ihren verbliebenen Ladungen absteigend sortiert sind.

Nun prüfe ich, ob N_{id} leer ist. Das ist der terminierende Fall in meinem Algorithmus. Wenn ja, prüfe ich, ob $T_{id} > 1$. Wenn die Batterie id mindestens zwei batteriefreie Felder besitzt, kann ich die restliche, aktuelle Ladung auf diesen beiden Feldern verbrauchen, indem wir einfach hin und her von einem Feld zum anderen übergehen. Dann erfolgt eine Prüfung, ob das Array C an allen Stellen 0 enthält. Wenn ja, ist R unser Ergebnis und der boolsche Wert 1 wird ausgegeben. Andernfalls haben wir kein Ergebnis und der boolsche Wert 0 wird ausgegeben.

Wenn N_{id} leer ist, prüfe ich auch, ob $T_{id} = 1$. Wenn ja, können wir die ein restlicher Schritt verbrauchen, indem wir zu einem batteriefreien Feld übergehen. Danach prüfen wir, ob das Array C an allen Stellen 0 enthält und die aktuelle Ladung 0 beträgt. Wenn ja, ist unser Ergebnis ebenfalls R . Der boolsche Wert 1 wird ausgegeben. Andernfalls wird der boolsche Wert 0 ausgegeben.

Anschließend, wenn N_{id} leer ist und es keine adjazenten batteriefreien Felder gibt, sammle ich alle erreichbaren Batterien von id aus. Ich sortiere sie absteigend nach Entfernung. Danach iteriere ich durch sie. Wenn es eine Entfernung gibt, deren Länge größer ist als 2, kann ich die ganze restliche aktuelle Ladung verbrauchen. Andernfalls kann ich nur so viel Ladung verbrauchen, wie lang die längste Entfernung ist. Danach wird geprüft, ob alle Ladungen in C 0 betragen und die aktuelle Ladung gleich 0 ist. Wenn ja, dann ist R unser Ergebnis und 1 wird ausgegeben. Andernfalls wird 0 ausgegeben.

Nun beschreibe ich, was man macht, wenn N_{id} nicht leer ist. Wir iterieren durch N_{id} . Jeden iterierten Nachbarn nenne ich i . Ich bilde ein Array D , in dem die möglichen Entfernungen, also nicht nur die minimale, gespeichert werden. Als erste Entfernung füge ich natürlich die minimale Entfernung von id zu i . Danach überprüfe ich, ob diese minimale Entfernung größer ist als 2. Wenn ja, füge ich in D die Summen von der minimalen Entfernung und vom nächsten Vielfachen von 2 ein, so lange eine solche Summe nicht größer ist als die aktuelle Ladung. Wenn nicht, prüfe ich den Wert $A'_{id,i}$, ob er mindestens 3 beträgt. Wenn ja, füge ich in D die Summen von $A'_{id,i}$ und vom nächsten Vielfachen von 2 ein, so lange eine solche Summe nicht größer ist als die aktuelle Ladung.

Danach iteriere ich durch D . Jede iterierte Entfernung nenne ich hier j . Ich speichere den Wert an der Stelle i im Array C als l . Ich kopiere C als C' und R als R' . Im Array C' speichere ich an der Stelle i die Differenz von der aktuellen Ladung und j . Ich füge i mit der Anzahl von Schritten j in R' ein. Nun lasse ich die rekursive Funktion mit folgenden Parametern laufen: dem Eingabeindex, l als neue aktuelle Ladung, das Array C' und das Array R' .

Wenn der ausgegebene boolsche Wert 1 ist, wird auch 1 ausgegeben und die Funktion bricht ab. Wenn nicht, wird das letzte Element aus dem Array D entfernt und die Entfernung $j + 1$ wird betrachtet.

Ganz am Anfang ist die erste aktuelle Ladung natürlich die Ladung der Batterie s , das heißt, der Startbatterie. An dieser Stelle setzen wir im Array C an der Stelle 0 den Wert 0. Auch wird 0 am Ende des Arrays R angehängt. Danach lassen wir unsere rekursive Funktion mit folgenden Parametern laufen: 0 als Eingabeindex, die aktuelle Ladung der Batterie s , das Array C und das Array R . Wenn der ausgegebene boolsche Wert 1 beträgt, wird die Folge von Schritten im gefundenen Pfad ausgegeben. Andernfalls erfolgt eine Meldung, dass kein solcher Pfad gefunden wurde.

1.4 Laufzeit

b – Anzahl der Batterien

l – Länge und Breite der Matrix M

Vorbereitung der Tabellen A und A' :

- Einlesen der Batterien: $O(b)$
- Bildung des Graphen: $O(l^2)$
Zuordnung der Brettindizes und Erstellung der Menge E : $O(l^2)$.

- Breitensuche: $O(b * l)$
Die Laufzeit vom Breitensuchealgorithmus ist $O(V + E)$, [1] $V = l$ und $E \leq 4 * l$: $O(l + l)$.
Der Algorithmus wird auf jeder Batterie angewendet: $O(b * l)$.
- Bestimmung der Schleifen in Knoten: $O(b)$
Für jede Batterie werden die Schleifen in Knoten bestimmt, eine solche Bestimmung läuft in $O(1)$,
also für alle Batterien: $O(b)$.

Wir addieren zusammen:

$$O(b) + O(l^2) + O(b * l) + O(b) = O(l^2 + b * l + 2b) \in O(l^2 + b * l)$$

Backtracking:

-

1.5 Generierung von Spielsituationen

Beim Lösen dieses Problems lesen wir die Koordinaten der Batterien ein, bestimmen die Entfernungen zwischen ihnen, lassen den rekursiven Algorithmus laufen und am Ende bekommen wir einen Pfad aus der Wurzel in s mit einer Folge von Schritten zwischen Batterien. Beim Generieren von solchen Spielsituationen gehen genau andersherum vor. Wir generieren zunächst die Folge von Schritten, die zwischen Batterien zu machen sind, sodass die ganze Spielsituation lösbar ist.

Um das Programm unter vielen Umständen nutzbar zu machen, lasse ich den Benutzer eine Anzahl von Batterien n einzugeben. Bei der Generierung einer Folge von Schritten unterscheide ich zwischen zwei Situationen:

- *Hinzufügen* – in dieser Situation sind Batterien a und b so platziert, dass wir von a zu b so übergehen, dass die Ladung, die wir bei a finden reicht, um b ohne Verlust an der Ladung zu erreichen. Es ist zu beachten, dass es nicht bedeutet, dass diese Ladung genau der minimalen Entfernung d_{min} von a zu b entspricht. Es kann sich um eine Entfernung $d_{min} + 2k$, wobei $k \in \mathbb{N}^+$, handeln.
- *Einfügen* – in dieser Situation handelt es sich darum, dass wir von einer Batterie a zu b übergehen und müssen danach zu

TODO: Einfügen ist schwierig und counterintuitive

1.5.1 Laufzeit

1.6 Erweiterungen

Literatur

- [1] T.H. Cormen u.a. *Introduction To Algorithms. Third edition*. Introduction to Algorithms. MIT Press, 2009, S. 597. ISBN: 9780262533058.

2 Umsetzung

In meinem Programm wird eine Batterie als eine Klasse **Battery** dargestellt. Jedes Objekt einer solchen Klasse besitzt jeweils eine x - und y -Koordinate, einen ganzzahligen Wert **charge**, der der Ladung einer Batterie entspricht; einen Brettindex **boardID** und einen Eingabeindex **inputID**.

2.1 Die Klasse Graph

Wir erstellen zwei Matrizen: **distances** und **distancesAux**, die den Matrizen A und A' entsprechen. Sie sind jeweils auf folgender Weise erstellt: **vector< vector<int> >**. Außerdem erstellen wir noch ein Array **extraTiles**, das dem Array T entspricht.

In dieser Klasse wird die Textdatei eingelesen. Mit Hilfe der eingelesenen Größe der Matrix M **boardDimension**, erstelle ich eine Matrix als **vector< vector<int> >**, die ich **board** nenne. Beim Einlesen jeder Batterie werden ihr die eingelesenen Koordinaten, sowie die Ladung zugewiesen. Danach werden jeder Batterie ihre Indizes zugeordnet. Als **inputID** gilt die Reihenfolge, in der die Batterien in der Textdatei auftreten. Die Startbatterie s bekommt einen Eingabeindex von 0. Die restlichen Batterien bekommen entsprechend die Indizes um 1 größer als ihre Vorgänger. Für die Bestimmung des Brettindex einer Batterie b bediene ich mich der folgenden Formel:

$$\text{boardID}(b) = (b_y - 1) * \text{boardDimension} + b_x - 1$$

Danach füge ich jeweilige Batterie in einen Map-Container **batteryToBoardID** mit ihrem entsprechenden Brettindex und gleich danach in einen Map-Container **batteryToBoardID** ein. Im ersten dienen die Batterien als Schlüssel und werden ihnen Brettindizes zugeordnet und im zweiten passiert das Gleiche aber andersherum: mit Brettindizes als Schlüsseln.

Danach erfolgt das Gleiche, wir fügen jeweilige Batterie in Map-Container **batteryToInputID** und **inputIDToBattery** ein, diesmal mit ihren entsprechenden Eingabeindizes.

Es wird ein Array **nodes** in Form von **vector< vector< pair<int, int> > >** erstellt, das die Nachbarn jedes Brettindex in der Matrix in Form von **pair**(Brettindex des Nachbarn, die Ladung des Nachbarn) enthält.

In der Methode **determineConnections()** wird die Matrix **board** iteriert. Jedem Brettindex werden ihre Nachbarn im Array **neighbors** gespeichert. Bei jeder Iteration werden die Brettindizes der Nachbarn anhand der obenstehenden Formel bestimmt. Die Ladung an einer Stelle rufen wir aus der Matrix **board** ab. Wir speichern die beiden Informationen im genannten Array. So durchlaufen wir alle Nachbarn jeder Stelle in der Matrix M . Am Ende fügen wir das Array **neighbors** am Ende des Arrays **nodes**.

In der Methode **BFS(Battery b)** läuft natürlich der Breitensuche-Algorithmus. Wir erstellen zwei lokale Arrays **battDistances** und **battDistancesAux** jeweils der Länge der Anzahl aller Batterien. Außerdem erstellen wir ein Array **vis** mit **bool** der Größe **boardDimension²**, in dem wir die besuchten Knoten markieren.

Als **currInputID** speichern wir den Eingabeindex der Batterie **b**. Wir formen eine Warteschlange **q**, die aus **pair<int, int>** besteht. Jedes solche Paar enthält den Brettindex und Entfernung in Schritten von der Batterie **b**.

Wir fügen in **q** den Brettindex von **b** mit der Entfernung 0 ein. Wir markieren im Array **vis** den entsprechenden Brettindex mit 1. Dann folgt die Iteration der Warteschlange, die so lange dauert, bis es noch Elemente in **q** gibt.

Als **currBoardID** speichern wir den Brettindex, der sich am Anfang der Warteschlange befindet und als **currDist** speichern wir die Entfernung des Feldes **currBoardID** von **q**. Sofort entfernen wir auch dieses erste Element aus der Warteschlange. Es folgt eine Iteration durch die Nachbarn von **currBoardID** im Array **nodes**.

Als **neighBoardID** und **neighCharge** bezeichne ich entsprechend den Brettindex des iterierten Nachbarn und die auf seinem Feld liegende Ladung. Es wird zuerst überprüft, ob **neighBoardID** bereits besucht wurde. Wenn ja, wird der nächste Nachbar genommen.

Dann wird überprüft, ob **neighCharge** größer als 0 ist, das heißt, ob auf dem Feld **neighBoardID** eine Batterie liegt.

Wenn nicht, wird **neighBoardID** mit dem Wert **currDist + 1** in **q** eingefügt. Auch wird **neighBoardID**

in **vis** mit 1 gekennzeichnet.

Wenn ja, wird die entsprechende Batterie anhand **neighBoardID** im Map-Container **boardIDToBattery** gefunden. Die Laufzeit der Suchfunktion des Map-Containers wird im Abschnitt 1.4 nicht betrachtet. Die gefundene Batterie nenne ich **neighborB**. Ihren Eingabeindex nenne ich **currNeighInputID**. Nun wenn **battDistances** an der Stelle **currNeighInputID** gleich 2 oder 1 ist, wird der Wert in **battDistancesAux** an der Stelle **currNeighInputID** aktualisiert, wenn er größer ist als **currDist + 1**, oder wenn noch keinen solchen Wert gibt, wird als **currDist + 1** gespeichert.

Andernfalls wird in **battDistances** an der Stelle **currNeighInputID** der Wert **currDist + 1** gespeichert. Nun nur wenn **currDist + 1** größer ist als 2, wird **neighBoardID** als besucht in **vis** gekennzeichnet.

Wenn es keine weiteren Brettindizes in der Warteschlange gibt, füge ich das Array **battDistances** an der Stelle von **currInputID** in **distances** ein. Das Gleiche erfolgt für das Array **battDistancesAux**. Es wird in **distancesAux** an der Stelle **currInputID** gespeichert.

Die Methode **checkOneTile(Battery b)** prüft, ob es sich neben dem Feld, auf dem die Batterie **b** liegt, ein batteriefreies Feld befindet.

Es wird das Array **nodes** an der Stelle, die dem Brettindex von **b** entspricht, iteriert. Es wird überprüft, ob mindestens ein Nachbar von **b** eine Ladung von 0 besitzt, das heißt, auf diesem Feld keine Batterie liegt. Es wird 1 ausgegeben, falls es ein Feld gibt, auf dem keine Batterie liegt. Andernfalls wird 0 ausgegeben.

Die Methode **checkTwoTiles(Battery b)** prüft, ob sich neben dem Feld, auf dem die Batterie **b** liegt, ein batteriefreies Feld befindet und dann überprüft, ob es noch ein batteriefreies Feld neben diesem Feld gibt.

Diese Funktion funktioniert auf ähnlicher Weise wie die Methode **checkOneTile**. Nun iterieren wir noch durch das Array von Nachbarn vom batteriefreien Nachbarn von **b** in **nodes**. Wenn es ein solches batteriefreie Feld gibt, wird 1 ausgegeben. Andernfalls wird 0 ausgegeben.

Im Konstruktor dieser Klasse lassen wir die Methoden **readFile** und dann **determineConnections** laufen. Danach für jede Batterie lassen wir die Methode **BFS** laufen. Gleich danach wenden wir die Methode **checkOneTile** an jeweiliger Batterie an und wenn 1 ausgegeben wird, schreiben wir 1 an der Stelle des Eingabeindex dieser Batterie in **extraTiles**. Danach wenden wir die Methode **checkTwoTiles** an jeweiliger Batterie an und wenn 1 ausgegeben wird, schreiben wir 2 an der Stelle des Eingabeindex dieser Batterie in **extraTiles**.

So bekommen wir eine Tabelle **distances** mit allen minimalen Entfernungen von jeder Batterie zu allen anderen. Die Entfernungen zu den Batterien, die nicht erreicht werden können, betragen 0. Außer diesen Batterien besitzt nur die Batterie, von der wir die Entfernungen messen, den Wert 0. Für Schleifen haben wir ja auch das Array **extraTiles**. Das bedeutet, dass wir in weiteren Betrachtungen die Stellen, an denen 0 steht, überhaupt nicht betrachten müssen.

2.2 Die Klasse Backtracking

In dieser Klasse werden die Arrays **distances**, **extraTiles** und **distancesAux** aus der Klasse **Graph** abgerufen und als **dist**, **extraTiles** und **distAux** gespeichert. Die Startbatterie wird als **start** gespeichert. Außerdem werden die Map-Container **batteryToID** und **IDToBattery** aus der Klasse **Graph** abgerufen, um die Koordinaten der Batterien zuordnen zu können.

Anschließend befindet sich in dieser Klasse ein Array **foundPath**, das die Folge von Schritten speichert, wenn eine Spielsituation lösbar ist.

In dieser Klasse befinden sich auch zwei Methoden: **checkReachability** und **next**.

Die erste Methode nimmt als Parameter einen Eingabeindex **id**, eine aktuelle Ladung **charge** und eine Kopie des Arrays **C** **currCharges**, also ein Array, das die Ladungen jeder Batterie enthält.

Es werden ein Array **reach** in Form von **vector<pair<int, int> >** und ein Array **poss** in Form von **vector<tuple<int, int, int> >** gebildet. Die beiden haben die gleiche Funktion, aber das zweite wird benutzt, um die Daten auf das erste vorzubereiten.

Es wird die Tabelle **dist** an der Stelle **id** iteriert. Jeden iterierten Eingabeindex einer Batterie nenne ich hier **i**. Wenn **i** und **id** gleich sind, prüfe ich, ob **extraTiles** an der Stelle **id** größer ist als 0 und ob **charge** mindestens 2 beträgt. Wenn ja, werden **i** und **extraTiles** an der Stelle **id** am Ende des Arrays **reach** angehängt.

Andernfalls, bei allen anderen Batterien, wird geprüft, ob der Wert an der Stelle (**id**, **i**) in der Tabelle

dist nicht größer ist als **charge**, ob der Wert an der Stelle (**id**, **i**) in der Tabelle **dist** nicht 0 beträgt und ob die Ladung an der Stelle **i** im Array **currCharges** nicht 0 ist. Wenn alle diese Bedingungen gleichzeitig erfüllt sind, hänge ich den Wert an der Stelle **i** im Array **currCharges**, **i** und den Wert an der Stelle (**id**, **i**) im Array **dist** am Ende des Arrays **poss** an.

Danach sortiere ich absteigend nach den Werten von **currCharges** im Array **poss**. Anschließend vervollständige ich das Array **reach** mit den sortierten Werten **i** und den Werten aus **dist**. Dieses fertige Array wird am Ende ausgegeben.

Nun kommen wir zur rekursiven Funktion **next**, die folgende Parameter nimmt: einen Eingabeindex **id**, eine aktuelle Ladung **charge**, eine Kopie vom Array **C** **status** und eine Kopie des Arrays **R** mit dem Ergebnispfad **result**.

Wir beginnen mit einem Lauf der Methode **checkReachability**, die die von der Batterie **id** aus erreichbaren Batterien findet. Diese Liste speichern wir unter **neighbors**. Nun prüfen wir, ob diese Liste leer ist. Wenn ja, haben wir folgende Fallunterscheidungen.

Wenn **charge** gleich 0 ist, prüfen wir im Array **status**, ob alle Batterien entladen sind. Wenn ja, wird **foundPath** als **result** gespeichert und ein boolescher Wert 1 ausgegeben. Andernfalls wird 0 ausgegeben. Wenn der Wert an der Stelle **id** im Array **extraTiles** größer ist als 2, wird am Ende des Arrays **result** -1 mit der restlichen aktuellen Ladung angehängt. Danach wird geprüft, ob alle Ladungen im Array **status** 0 betragen. Wenn ja, wird **foundPath** als **result** gespeichert und ein boolescher Wert 1 ausgegeben. Andernfalls wird 0 ausgegeben.

Wenn der Wert an der Stelle **id** im Array **extraTiles** gleich 1 ist, wird am Ende des Arrays **result** -1 mit dem Wert 1 angehängt. **charge** wird um 1 verkleinert. Danach wird geprüft, ob alle Ladungen im Array **status** 0 betragen und ob **charge** gleich 0 ist. Wenn ja, wird **foundPath** als **result** gespeichert und ein boolescher Wert 1 ausgegeben. Andernfalls wird 0 ausgegeben.

In allen anderen Fällen wird nun ein Array **allNeighbors** gebildet. In das Array füge ich alle Eingabeindizes und alle minimale Entfernungen der Batterien, die erreichbar sind, unabhängig davon, ob sie entladen sind oder nicht. Ich sortiere dieses Array absteigend nach den minimalen Entfernungen. Danach iteriere ich durch das Array **allNeighbors** und wenn ich auf eine minimale Entfernung komme, die nicht kleiner ist als 3, hänge ich am Ende des Arrays **result** ihren entsprechenden Eingabeindex mit der restlichen aktuellen Ladung an. Gleichzeitig wird die Iteration abgebrochen. Andernfalls, wenn ich auf eine minimale Entfernung komme, die kleiner ist als 3, hänge ich am Ende des Arrays **result** ihren entsprechenden Eingabeindex mit dieser minimalen Länge an. Von **charge** wird der Wert von dieser minimalen Entfernung abgezogen. Gleichzeitig wird die Iteration abgebrochen.

Danach wird geprüft, ob alle Ladungen im Array **status** 0 betragen und ob **charge** gleich 0 ist. Wenn ja, wird **foundPath** als **result** gespeichert und ein boolescher Wert 1 ausgegeben. Andernfalls wird 0 ausgegeben.

Nun kommen wir zum Fall, wenn **neighbors** nicht leer ist. Wir iterieren durch dieses Array. Unter **neighID** speichere ich den iterierten Eingabeindex und unter **minDist** speichere ich die iterierte minimale Entfernung. Ich bilde ein Array **neighDistances**, das alle Entfernungen zwischen **id** und **neighID** enthält. Ich füge zuerst in es **minDist** ein. Nun wird geprüft, ob **minDist** größer als 2 ist. Wenn ja, wird zu **minDist** 2 addiert und in **neighDistances** eingefügt, so lange diese neue Summe nicht größer ist als **charge**.

Wenn nicht, wird überprüft, ob der Wert an der Stelle (**id**, **neighID**) in der Tabelle **distAux** größer ist als 2. Wenn ja, wird zum Wert an der Stelle (**id**, **neighID**) in der Tabelle **distAux2** addiert und in **neighDistances** eingefügt, so lange diese neue Summe nicht größer ist als **charge**.

Danach erfolgt eine Iteration von **neighDistances**. Als **nextCharge** speichere ich den Wert im Array **status** an der Stelle **neighID**. Ich bilde eine Kopie von **status** und speichere sie als **cpstatus** und eine Kopie von **result** und speichere sie als **cpresult**. An der Stelle **neighID** im Array **cpstatus** speichere ich die Differenz von **charge** und der iterierten Entfernung. Am Ende des Arrays **cpresult** hänge ich **neighID** und die iterierte Entfernung. Danach lasse ich die Funktion **next()** rekursiv mit folgenden Parametern laufen: **neighID** als Eingabeindex, **nextCharge** als aktuelle Ladung, **cpstatus** als eine Kopie von **C** und **cpresult** als eine Kopie von **R**. Den ausgegebenen Wert speichere ich als **found**. Wenn **found** 1 beträgt gebe ich 1 aus. Andernfalls wird das letzte Element aus dem Array **cpresult** entfernt.

Bei der Deklaration dieser Klassen werden, wie beschrieben, die erwähnten Matrizen und Arrays aus der Klasse **Graph** abgerufen. Ein Array **startStatus** wird gebildet, in dem die Ladungen jeder Batterie gespeichert werden. Als **startCharge** speichere ich die erste aktuelle Ladung der Batterie **start**. An der Stelle 0 im Array **startStatus** wird 0 gesetzt. Ich bilde ein Array **results**, das dem Array **R**

entspricht. Ich hänge den Starteingabeindex und den Wert 0 am Ende dieses Arrays an. Dann lasse ich die Funktion `next` mit folgenden Parametern laufen: den Eingabeindex der Startbatterie, `startCharge` als die aktuelle Ladung, `startStatus` und `results`. Zwischen dem Start dieser Operation und ihr Ende messe ich die vergangene Zeit mit Hilfe von Namespace `steady_clock`. Wenn ein Pfad von der Startbatterie gefunden wird, wird er in Form von den Elementen von `foundPath` ausgegeben. Wenn nicht, erfolgt eine entsprechende Meldung. Anschließend wird das Format der Zeit so angepasst, dass nur die Minuten und Sekunden angezeigt werden.

2.3 Die Klasse Generator

3 Beispiele

Die untenstehenden Zahlen, die die Folgen der Schritten bei einer Spielsituation darstellen, stehen für die Eingabeindizes der Batterien in einer Spielsituation. Diese Indizes entsprechen der Reihenfolge der Batterien in der Textdatei. Man fängt mit 0 an, diese Zahl entspricht der Startbatterie.

Eine Notation $a(d_a) \rightarrow b(d_b)$ bedeutet, dass man von der Batterie mit dem Eingabeindex a zur Batterie b genau d_b Schritten gemacht hat. Das heißt, dass man von der Ladung von a d_b abgezogen hat und die übrige Ladung auf das Feld mit b gestellt hat.

Die Zahl -1 steht für ein beliebiges Feld, zu dem man am Ende übergeht, um die restliche Ladung aus der letzten Batterie auszunutzen.

3.1 Beispiel 0 (BWINF)

Textdatei: stromrallye0.txt

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 3(3) \rightarrow 1(3) \rightarrow 3(3) \rightarrow 2(6) \rightarrow 2(2)$

3.2 Beispiel 1 (BWINF)

Textdatei: stromrallye1.txt

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 1(1) \rightarrow 2(1) \rightarrow 3(1) \rightarrow 4(1) \rightarrow 5(1) \rightarrow 6(1) \rightarrow 7(1) \rightarrow 8(1) \rightarrow 9(1) \rightarrow 19(1) \rightarrow 18(1) \rightarrow 17(1) \rightarrow 16(1) \rightarrow 15(1) \rightarrow 14(1) \rightarrow 13(1) \rightarrow 12(1) \rightarrow 11(1) \rightarrow 10(1) \rightarrow 20(1) \rightarrow 21(1) \rightarrow 22(1) \rightarrow 23(1) \rightarrow 24(1) \rightarrow 25(1) \rightarrow 26(1) \rightarrow 27(1) \rightarrow 28(1) \rightarrow 29(1) \rightarrow 39(1) \rightarrow 38(1) \rightarrow 37(1) \rightarrow 36(1) \rightarrow 35(1) \rightarrow 34(1) \rightarrow 33(1) \rightarrow 32(1) \rightarrow 31(1) \rightarrow 30(1) \rightarrow 40(1) \rightarrow 41(1) \rightarrow 42(1) \rightarrow 43(1) \rightarrow 44(1) \rightarrow 45(1) \rightarrow 46(1) \rightarrow 47(1) \rightarrow 48(1) \rightarrow 49(1) \rightarrow 59(1) \rightarrow 58(1) \rightarrow 57(1) \rightarrow 56(1) \rightarrow 55(1) \rightarrow 54(1) \rightarrow 53(1) \rightarrow 52(1) \rightarrow 51(1) \rightarrow 50(1) \rightarrow 60(1) \rightarrow 61(1) \rightarrow 62(1) \rightarrow 63(1) \rightarrow 64(1) \rightarrow 65(1) \rightarrow 66(1) \rightarrow 67(1) \rightarrow 68(1) \rightarrow 69(1) \rightarrow 79(1) \rightarrow 78(1) \rightarrow 77(1) \rightarrow 76(1) \rightarrow 75(1) \rightarrow 74(1) \rightarrow 73(1) \rightarrow 72(1) \rightarrow 71(1) \rightarrow 70(1) \rightarrow 80(1) \rightarrow 81(1) \rightarrow 82(1) \rightarrow 83(1) \rightarrow 84(1) \rightarrow 85(1) \rightarrow 86(1) \rightarrow 87(1) \rightarrow 88(1) \rightarrow 89(1) \rightarrow 99(1) \rightarrow 98(1) \rightarrow 97(1) \rightarrow 96(1) \rightarrow 95(1) \rightarrow 94(1) \rightarrow 93(1) \rightarrow 92(1) \rightarrow 91(1) \rightarrow 90(1) \rightarrow 80(1) \rightarrow 91(1)$

3.3 Beispiel 2 (BWINF)

Textdatei: stromrallye2.txt

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 50(1) \rightarrow 39(1) \rightarrow 28(1) \rightarrow 17(1) \rightarrow 6(1) \rightarrow 5(1) \rightarrow 4(1) \rightarrow 3(1) \rightarrow 2(1) \rightarrow 1(1) \rightarrow 12(1) \rightarrow 13(1) \rightarrow 14(1) \rightarrow 15(1) \rightarrow 16(1) \rightarrow 27(1) \rightarrow 26(1) \rightarrow 25(1) \rightarrow 24(1) \rightarrow 23(1) \rightarrow 34(1) \rightarrow 35(1) \rightarrow 36(1) \rightarrow 37(1) \rightarrow 38(1) \rightarrow 49(1) \rightarrow 48(1) \rightarrow 47(1) \rightarrow 46(1) \rightarrow 45(1) \rightarrow 56(1) \rightarrow 57(1) \rightarrow 58(1) \rightarrow 59(1) \rightarrow 60(1) \rightarrow 70(1) \rightarrow 69(1) \rightarrow 68(1) \rightarrow 67(1) \rightarrow 66(1) \rightarrow 77(1) \rightarrow 78(1) \rightarrow 79(1) \rightarrow 80(1) \rightarrow 81(1) \rightarrow 82(1) \rightarrow 71(1) \rightarrow 72(1) \rightarrow 61(1) \rightarrow 51(1) \rightarrow 40(1) \rightarrow 29(1) \rightarrow 18(1) \rightarrow 7(1) \rightarrow 8(1) \rightarrow 9(1) \rightarrow 10(1) \rightarrow 11(1) \rightarrow 22(1) \rightarrow 21(1) \rightarrow 20(1) \rightarrow 19(1) \rightarrow 30(1) \rightarrow 31(1) \rightarrow 32(1) \rightarrow 33(1) \rightarrow 44(1) \rightarrow 43(1) \rightarrow 42(1) \rightarrow 41(1) \rightarrow 52(1) \rightarrow 53(1) \rightarrow 54(1) \rightarrow 55(1) \rightarrow 65(1) \rightarrow 64(1) \rightarrow 63(1) \rightarrow 62(1) \rightarrow 73(1) \rightarrow 74(1) \rightarrow 75(1) \rightarrow 76(1) \rightarrow 87(1) \rightarrow 86(1) \rightarrow 85(1) \rightarrow 84(1) \rightarrow 83(1) \rightarrow 94(1) \rightarrow 93(1) \rightarrow 92(1) \rightarrow 91(1) \rightarrow 90(1) \rightarrow 89(1) \rightarrow 88(1) \rightarrow 99(1) \rightarrow 100(1) \rightarrow 101(1) \rightarrow 102(1) \rightarrow 103(1) \rightarrow 104(1) \rightarrow 105(1) \rightarrow 106(1) \rightarrow 95(1) \rightarrow 96(1) \rightarrow 97(1) \rightarrow 98(1) \rightarrow 109(1) \rightarrow 108(1) \rightarrow 107(1) \rightarrow 118(1) \rightarrow 117(1) \rightarrow 116(1) \rightarrow 115(1) \rightarrow 114(1) \rightarrow 113(1) \rightarrow 112(1) \rightarrow 111(1) \rightarrow 110(1) \rightarrow 99(1) \rightarrow 88(1) \rightarrow 77(1) \rightarrow 66(1) \rightarrow 56(1) \rightarrow 45(1) \rightarrow 34(1) \rightarrow 23(1) \rightarrow 12(1) \rightarrow 1(1) \rightarrow 2(1) \rightarrow 3(1) \rightarrow 4(1) \rightarrow 5(1) \rightarrow 6(1) \rightarrow 7(1) \rightarrow$

8(1) → 9(1) → 10(1) → 11(1) → 22(1) → 21(1) → 20(1) → 19(1) → 18(1) → 17(1) → 16(1) → 15(1) → 14(1) → 13(1) → 24(1) → 25(1) → 26(1) → 27(1) → 28(1) → 29(1) → 30(1) → 31(1) → 32(1) → 33(1) → 44(1) → 43(1) → 42(1) → 41(1) → 40(1) → 39(1) → 38(1) → 37(1) → 36(1) → 35(1) → 46(1) → 47(1) → 48(1) → 49(1) → 50(1) → 51(1) → 52(1) → 53(1) → 54(1) → 55(1) → 65(1) → 64(1) → 63(1) → 62(1) → 61(1) → 72(1) → 71(1) → 70(1) → 60(1) → 59(1) → 58(1) → 57(1) → 67(1) → 68(1) → 69(1) → 80(1) → 79(1) → 78(1) → 89(1) → 90(1) → 91(1) → 92(1) → 81(1) → 82(1) → 83(1) → 84(1) → 73(1) → 74(1) → 75(1) → 76(1) → 87(1) → 86(1) → 85(1) → 96(1) → 95(1) → 94(1) → 93(1) → 104(1) → 103(1) → 102(1) → 101(1) → 100(1) → 99(1) → 110(1) → 111(1) → 112(1) → 113(1) → 114(1) → 115(1) → 116(1) → 105(1) → 106(1) → 107(1) → 108(1) → 97(1) → 98(1) → 109(1) → 120(1) → 119(1) → 118(1) → 117(1) → 118(1) → 119(1) → 108(1) → 118(1) → 120(1)

3.4 Beispiel 3 (BWINF)

Textdatei: stromrallye3.txt

Die Spielsituation ist nicht lösbar.

Zeit: 0 min 0 s

3.5 Beispiel 4 (BWINF)

Textdatei: stromrallye4.txt

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow -1(20)$

3.6 Beispiel 5 (BWINF)

Textdatei: stromrallye5.txt

Die Spielsituation ist nicht lösbar.

Zeit: 16 min 59 s

3.7 Beispiel 6

Textdatei: stromrallye6.txt

Besonderheit: eine Kopie des Beispiels 3.4, aber die Batterie auf dem Feld (6, 2) besitzt eine Ladung von 4

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 2(9) \rightarrow 1(9) \rightarrow 3(2) \rightarrow 3(2) \rightarrow 3(2) \rightarrow 1(2) \rightarrow -1(1)$

3.8 Beispiel 7

Textdatei: stromrallye7.txt

Besonderheit: eine Kopie des Beispiels 3.6 mit einer zusätzlichen Batterie auf dem Feld (2, 16) mit einer Ladung von 1

Die Spielsituation ist lösbar.

Zeit: 9 min 16 s

$0 \rightarrow 1(4) \rightarrow 3(7) \rightarrow 4(2) \rightarrow 3(10) \rightarrow 2(3) \rightarrow 1(4) \rightarrow 5(6) \rightarrow 5(2) \rightarrow 12(10) \rightarrow 6(3) \rightarrow 6(2) \rightarrow 7(2) \rightarrow$
 $8(2) \rightarrow 9(2) \rightarrow 10(2) \rightarrow 11(2) \rightarrow 13(1) \rightarrow 14(1) \rightarrow 15(1) \rightarrow 16(1) \rightarrow 17(1) \rightarrow 18(1) \rightarrow 19(1) \rightarrow 20(3) \rightarrow$
 $21(1) \rightarrow 22(1) \rightarrow 23(1) \rightarrow 24(1) \rightarrow 25(1) \rightarrow 28(1) \rightarrow 27(1) \rightarrow 26(1) \rightarrow 27(1) \rightarrow 30(1) \rightarrow 31(1) \rightarrow$
 $30(1) \rightarrow 33(1) \rightarrow 34(1) \rightarrow 35(1) \rightarrow 32(1) \rightarrow 29(1) \rightarrow -1(1)$

3.9 Beispiel 8

Textdatei: `stromrallye8.txt`

Besonderheit: ein anhand des selbst gebauten Generators generiertes Beispiel

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 3(2) \rightarrow 1(3) \rightarrow 1(2) \rightarrow 2(3) \rightarrow 4(6) \rightarrow 5(4) \rightarrow 5(2) \rightarrow 5(2) \rightarrow 5(2) \rightarrow 5(2) \rightarrow 5(2) \rightarrow 5(2) \rightarrow$
 $5(2)$

3.10 Beispiel 9

Textdatei: `stromrallye9.txt`

Besonderheit: ein anhand des selbst gebauten Generators generiertes Beispiel

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \rightarrow 4(5) \rightarrow 3(5) \rightarrow 6(7) \rightarrow 8(6) \rightarrow 9(3) \rightarrow 8(3) \rightarrow 8(2) \rightarrow 9(5) \rightarrow 6(7) \rightarrow 7(5) \rightarrow 2(6) \rightarrow 1(2) \rightarrow 3(4) \rightarrow$
 $5(1) \rightarrow 5(2) \rightarrow 4(4) \rightarrow -1(1)$

4 Quellcode

stromrallye.m