Aufgabe 1: Stromrallye

Teilnahme-Id: 52586

Bearbeiter dieser Aufgabe:
Michal Boron

April 2020

Inhaltsverzeichnis

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 $x_i \leqslant l$ und $y_i \leqslant l$ 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 Schitte entpricht, 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 serach, BFS*) einfach bestimmen lassen.

Für eine Batterie b können wir auf folgender Weise die Entfernungen zu allen anderen Batterien bestimmen. Sie werden in einer zweidimensionaller 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 nich 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 benahcbarten 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 präsentiert ein Beispiel, in dem diese besondere Unterscheidung wichtig ist.

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

5	
1	

5	2
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.

wir die Schritte: $b \to i \to j \to i \to b$. Auch hier können wir die in der Beobachtung 3 bemerkte Ordnungsgemäßheit anwenden, wenn wir l statt $A_{i,j}$ nehmen. (s. Abb. 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 zusätliche, 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 $2n$ Schitte 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 ereichbar 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 zusätlichen, 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 bermeken, 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 entstanden 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. Daraux 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 mih 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 Schitten zu erreichen, lässt man diese rekursive Funktion dann an der Batterie b laufen. Wenn es nicht geht, an einer Stelle fortzusetzten, wird abgebrochen und man geht zürück zu der Batterie, an der noch eine Möglichkeit entsteht, von dieser Batterie einen anderen Weg zu nehmen und mit einer anderen Anzahl von Schitten 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 serach, 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.

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 dadruch 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 ich 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

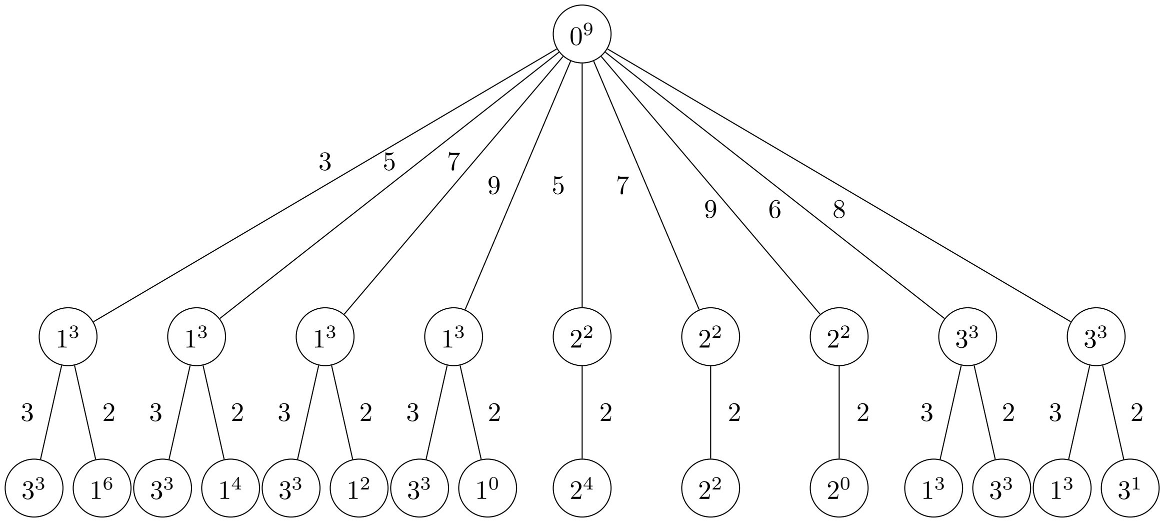

Abbildung 3: Ein abgebildeter Baum der 0., 1. und 2. Niveaus des Beispiels 3.1 (das Beispiel aus der Aufgabenstellung). Die Nummern in Knoten entsprechen den Eingabeindizes der Batterien. Die Potenzen stellen die Ladung dar, die wir auf einem Feld finden, wenn wir eine Batterie erreichen.

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 mir folgenden Parametern laufen: dem Eingabeindex, l als neue aktuelle Ladung, das Array C' und das Array R'.
Wenn der augegebene 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 aussgegeben. Andernfalls erfolgt eine Meldung, dass kein solcher Pfad gefunden wurde.

1.4 Laufzeit

b – Anzahl der Batterien
l – Länge oder 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 Esrstellung der Menge E: $O(l^2)$.
- Breitensuche: $O(b * l)$
  Die Laufzeit vom Breitensuchealgorithmus ist $O(V + E)$,[1] $V = l$ und $E \leqslant 4 * l$: $O(l + l)$.
  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 (worst-case):
Um sie zu bestimmen, bediene ich mich des Faktes, dass ich einen Möglichkeitsbaum bilde. (Beispiel, s. Abb. 3)
Von jeder Batterie kann man zu n Batterien übergehen. Dieser Wert kann maximal $n = b$ betragen, wenn man zu allen Batterien übergehen kann. Auch kann ich jede von diesen n Batterien in maximal $m = \frac{1}{2} * max(x : x \in C)$ möglichen Anzahlen von Schritten erreichen. Das folgende Prdukt stellt dar, wie viele Verbindungen es von einem Knoten zu seinen Blättern in einem Baum maximal existieren kann: $n * m$. Nun müssen wir diesen Wert noch an alle Knoten in diesem Baum übertragen. Das heißt, dieses Produkt wird zur Basis einer Potenz. Der Exponent dieser Potenz entspricht maximal $s = \sum_{i=0}^{C} C_i$, wie das in den Beispielen 3.2 und 3.3 gut sichtbar ist. In worst-case ist dann die Laufzeit:

$$O((n * m)^s), \text{wobei } n = b.$$

Diese Lauzeit kann horrend scheinen, aber werfen einen Blick wir auf unsere Beispiele.
Im Beispiel 3.2 haben wir $b = 100$ Batterien. Von jeder Batterie kann man maximal $n = 4$ andere Batterien erreichen. Zwischen jedem Paar von Batterien kann man maximal einen Schritt machen, also $m = 1$. Die Summe aller Ladungen beträgt $s = 100$. Wir kommen auf die folgende Laufzeit: $O(4 * 1)^{100}$.
Beim Beispiel 3.3 haben wir eine ähnliche Sitaution: $b = 121$, $n = 4$, $m = 1$, $s = 2 * 121 = 242$.
So kommen wir auf die folgende Laufzeit: $O(4 * 1)^{242}$. Dazu kommt noch die eingebaute Heuristik, die die erreichbaren Batterien nach ihren Ladung absteigend sortiert. Sie bechleunigt sehr deutlich den Lauf des Programms.
In anderen Beispielen (außer dem Beispiel 3.6/3.8) läuft mein Programm auch sehr schnell, weil s sehr gering bleibt.

1.5 Erweiterungen

1.5.1 Schräge Übergänge

Im Abschnitt 1.1 beschrieb ich die Möglichkeiten, zu welchen Feldern man einen Schritt machen darf. Als eine Erweiterung der Aufgabenstellung erlaube ich in meinem Programm auch schräge Übergänge, die ebenfalls als ein Schritt gezählt werden. Angenommen, stehen wir auf einem Feld f mit Koordinaten (x_f, y_f), so dürfen wir auch zu folgenden Feldern einen Schritt machen:

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

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

Diese Methode wurde angewendet, um das Beispiel 3.4 zu lösen, das normalerweise unlösbar bleibt. Das Ergebnis dieses Versuchs ist im Beispiel 3.11 zu finden.

Literatur

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

1.6 Generierung von Spielsituationen

Beim Lösen dieses Problems lesen wir die Koordinaten der Batterien ein, bestimmen die Entfernungen zwischen ihnen, lassen den rekursiven Algorihtmus laufen und am Ende bekommen wir einen Pfad aus der Wurzel in s mit einer Folge von Schritten zwischen Baterrien. Beim Generieren von solchen Spielsituation gehen wir 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 das 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 zurück zu a kommen.

Die erste Sitaution ist einfach und sehr intuitiv. Die zweite ist deutlich schwieriger, weil man beim per-Hand-Lösen nachdenken muss, wie man die restliche Ladung von a behandelt. Eine solche Situation ist kontraintuitiv besonders, wenn mehrere Einfügen-Sitautionen nacheinander auftreten.
Ich bilde ein Array W, in dem ich die Reihenfolge der Batterien speichere. Ich fordere vom Benutzer meines Programms, dass er eine Anzahl der Batterien eingibt, die mindestens 1 ist. Aus diesem Grund kann ich sofort am Ende des Arrays W 0 anhängen, also der Eingabeindex der Startbatterie. Danach generiere ich $n-1$ booleschen Werte (0 – Hinzufügen, 1 – Einfügen), deren Verteilung der stetigen Gleichverteilung entspricht. Wenn es sich um eine Einfügen–Situation handelt, darf ich nicht vergessen, dass ich sicherstellen muss, dass der Index der Batterie, die es betrifft, auch danach in der Reihenfolge auftritt. Aus diesem Grund bediene ich mich eines Stapelspeichers. Ich iteriere danach durch die generierten Werte. Gleichzeitig setzte ich auch einen Iterator i, der auf den nächsten Index einer Batterie zeigt, also nach 0. Wenn ich auf eine Hinzufügen–Situation komme, hänge ich einfach den Wert i am Ende des Arrays W an und überprüfe, ob es Indizes in dem Stapelspeicher gibt. Wenn ja, sehe ich den ersten nach, ich füge ihn am Endes des Arrays W und kellere ihn aus.
Wenn ich auf eine Einfügen–Situation komme, hänge ich am Ende des Arrays i an und kellere i in den Stapelspeicher ein.
So komme ich auf eine Reihenfolge von Eingabeindizes, mit Hilfe deren ich sicherstelle, dass eine Spielsituation lösbar ist. (s. unten Abb. 4 mit einer Beispielreihenfolge)

$$\begin{array}{c} 0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 4 \longrightarrow 5 \\ H \to E \to E \to H \to E' \to E' \to H \to H \end{array}$$

Abbildung 4: Eine Beispielsituation. Die Nummern in der oberen Reihe sind Eingabeindizes in W. Die Buchstaben unten sind die entsprechenden Situationen. E' weist darauf hin, dass der entsprechende Eingabeindex schon früher auftrat und es bei ihm um eine Einfügen–Situation handelte.

Nun kommen wir zur Generierung der Entfernungen zwischen den jeweiligen Indizes in W. Ich verbinde diese Operation gleichzeitig mit der Generierung der Koordinaten für jede Batterie.
Ich bilde ein Array F, in dem ich diese Entfernungen zwischen Indizes in der Reihenfolge enthalten sind. Es ist zu beachten, dass F der Länge $|F| = |W|-1$ ist. Ich lege auch ein Array K fest, in dem die Koordinaten jeder Batterie gespeichert werden. Ich bilde ein Array vis, in dem die Batterien gekennzeichnet werden, für die Koordinaten bereits bestimmt wurden. Ich lege auch eine Menge $used$ fest, in der die bestimmten Koordinaten enthalten werden.
Wie beim Generieren der Reihenfolge muss beachtet werden, dass wir bei einer Einfügen–Situation die Entfernungen in F spiegeln. Aus dem Grund bediene ich mich auch eines Stapelspeichers, der ein Paar von ganzen Zahlen speichert: einen Index und eine Entfernung.
Wir iterieren durch W vom ersten bis zum vorletzen Element. Den Iterator nenne ich i.
Ich generiere eine ganze Zahl aus dem Bereich $[1, 5]$, die ich $dist$ nenne. Ich kopiere diese Zahl als $currDist$.
Ich überprüfe, ob der Index des ersten Elements im Stapelspeicher gleich v_{i+1} ist. Wenn ja, wird $currDist$ als die Entfernung des ersten Elements im Stapelspeicher gespeichert. Dann geprüft wird, ob v_i in vis

als besucht markiert wurde. Wenn nicht, werden für v_{i+1} Koordinaten bestimmt. $currDist$ ist dann die Entfernung zwischen v_i und v_{i+1}. Als $prev$ bezeichne ich die Koordinaten in K an der Stelle v_i. Diesen Punkt auf der Ebene nehme ich als Ausgangspunkt bei der Bestimmung der Koordinaten für v_{i+1}. Nun können wir bemerken, dass es genau $4 * currDist$ mögliche Positionen gibt, die sich in der Entfernung, die $currDist$ beträgt, von $prev$ befinden. Ich nutze diese Information aus und ordne jedem Paar von Koordinaten eine $coorID$ zu.

$$coorID(x,y) = \begin{cases} 0 * currDist + x - 1, & \text{wenn } x \geqslant 0 \text{ und } y \geqslant 0 \\ 1 * currDist + x - 1, & \text{wenn } x \geqslant 0 \text{ und } y < 0 \\ 2 * currDist + x - 1, & \text{wenn } x < 0 \text{ und } y \geqslant 0 \\ 3 * currDist + x - 1, & \text{wenn } x < 0 \text{ und } y < 0 \end{cases} \tag{1}$$

Nun generiere ich einen ganzzahligen Wert $nextID$ aus dem Bereich $[0, 4 * currDist)$. Dieser Generator folgt der stetigen Gleichverteilung. Anhand dieser generierten Zahl bestimme ich die x– und y–Koordinaten. Zunächst ohne möglichen Minuszeichen. Die y–Koordinate lässt sich einfach bestimmen: $y = currDist - |x|$. Nach Bedarf ordne ich dann die Minuszeichen nach der Bestimmung von Koordinaten x und y zu. Anschließend addiere ich zu x und y entsprechend die x– und y–Koordinaten von $prevPoint$.
Nun sollen wir bemerken, dass es zu einer Situation kommen kann, dass ein neu enstandenes Paar von Koordinaten schon benutzt wurde und einer anderen Batterie zugeordnet ist. Das prüfe ich mit Hilfe von $used$. Aus diesem Grund müssen wir einfach eine neue $coorID$ generieren. In einem Array $free$ kennzeichne ich jedes Mal die gerade generierte $coorID$ als bereits benutzt. Wenn es im Array $free$ Stellen gibt, an denen noch 0 steht, also das bedeutet, dass diese Kooridnatenkombination nicht benutzt wurde, haben wir noch ein mögliches Paar, das wir generieren können. Der Generator generiert die Zahlen unabhögig davon, ob sie in $free$ markiert wurden. Der Generator folgt jedoch der stetigen Gleichverteilung und eine Situation, in der wir jedes Mal die gleiche Zahl generieren ist minimal.
Da ich eine ganzzahlige Entfernung aus dem Bereich $[1, 5]$ generiere, kann es bei größeren n zu Situationen kommen, dass alle möglichen $coorID$ benutzt werden und man kann einer Batterie ein Paar von Koordinaten zuordnen, die sich in der Entfernung $nextDist$ von $prev$ befinden. So muss ich an dieser Stelle $currDist$ um 1 vergrößern und die ganze Operation von Bestimmung eines Koordinatenpaars läuft vom Anfang an. Wenn ich ein passendes Paar von Koordinaten finde, speichere ich $currDist$ als $dist$, markiere v_{i+1} in vis als besucht, füge speichere die Koordinaten an der Stelle v_{i+1} im Array $coor$ und füge dieses Parr in $used$ ein.
Anschließend müssen wir noch die gefundene Entfernung am Ende des Arrays F anhängen. Wir erinnern uns auch an die Spiegelung, die ich früher erwähnte. Wir müssen nun prüfen, ob der Index des ersten Elements im Stapelspeicher gleich v_{i+1} ist. Wenn ja, hängen wir die Entfernung, die beim ersten Element im Stapelspeicher steht, am Ende des Arrays F an. Es folgt eine Auskellerung des ersten Elements im Stapelspeicher.
Wenn nicht, hänge ich am Ende des Arrays F $dist$ an. Gleichzeitig kellere ich auch diesen Wert mit dem Index v_i in den Stapelspeicher ein. (Beispiel, s. Abb 5)
Nach der Bestimmung des Arrays F suche ich in $coor$ nach der kleinsten Koordinate von allen. Ich speichere sie als $minimal$. Danach multipliziere ich diese Variable mal -1 und addiere 2. Ich addiere $minimal$ zu allen Koordinaten in $coor$. Ich muss es tun, weil es Koordinaten geben kann, die negativ sind. So verschiebe ich die ganze Matrix um den Wert $minimal$ und ich stelle sicher, dass es keine Koordinate gibt, die negativ ist. Ich will auch sicher stellen, dass es bei den Batterien am Rand möglich ist eine Schleife durchzuführen. Aus dem Grund addiere ich 2. Mehr dazu im nächsten Abschnitt.

$$\begin{array}{c} 0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 4 \longrightarrow 5 \\ 2 \longrightarrow 3 \longrightarrow 1 \longrightarrow 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 2 \end{array}$$

Abbildung 5: Eine Erweiterung der Abb. 4. Die Zahlen in der unteren Reihenfolge stehen für die Entfernungen zwischen jeweiliger über ihnen stehenden Indizes, also die Zahlen in F. Die Farbung weist darauf hin, dass die zweite Entfernung von der ersten enifach gespiegelt, kopiert wurde, weil es sich an dieser Stelle um eine Einfügen–Situation handelte.

Danach kommen wir zur Generierung von Ladungen. Dazu lege ich ein Array H und ein Array P. Es ist zu beachten, dass die Ladung bei einer Batterie a mindestens so groß sein muss wie groß die Entfernung zwischen a und b ist, wenn es sich bei a um eine Hinzufügen–Situation handelt. Wemn es eine Einfügen–Situation ist, muss die Ladung bei einer Batterie a mindestens so groß sein wie die Entfernung von a zu

b und die Entfernung zwischen b und c zusammen. Also muss eine solche Ladung mindestens die Summe der Entfernungen zwischen a, b und b, c betragen.
Ich iteriere durch W. Jeden interierten Index nenne ich i. Ich überprüfe, ob der es sich an der Stelle W_i im Array H bereits einen Wert gibt. Wenn ja, beduetet es, dass wir mit einer Einfügen–Situation bei W_i zu tun hatten und ich addiere an der Stelle P_{W_i} im Array H den Wert F_i.
Andernfalls schreibe ich an dieser Stelle den Wert F_i. Außerdem speichere ich gleichzeitig an der Stelle W_i im Array P den Wert W_{i-1} (ich überprüfe davor, ob i nicht 0 beträgt).
Danach erfolgt eine Iteration vom Array H. Den Iterator nenne ich i. Ich will sicherstellen, dass jede Batterie mindestens eine Ladung besitzt und wenn eine Batterie H_i einen Wert 0 besitzt, als nur die letzte Batterie, generiere ich einen ganzzahligen Wert $mult$ aus dem Bereich $[0, 2]$. Dieser Generator folgt der stetiger Gleichverteilung. Zum Wert H_i addiere ich den Wert $2 * mult$ und addiere dann 1.
Wenn eine Batterie i schon einen Wert H_i besitzt, der nicht 0 ist, generiere ich ebenfalls einen ganzzahligen Wert $mult$ aus dem Bereich $[0, 2]$. Ich addiere zu H_i $2 * mult$.
Ich addiere diese zusätzlichen Vielfachen von 2, um die Spielsituation schwieriger zu machen, weil die Person, die die Aufgabe lösen muss, zusätzlich nachdenken muss, was sie mit der gebliebenen Ladung machen soll. Die zusätzliche Ladung verwirrt auch die Person, weil sie einen Eindruck gibt, dass es vielleicht einen anderen Pfad von der Batterie geben kann. Manche Personen können auch gar darauf kommen, dass man eine Schleife zu einem Feld machen kann oder dass man einen längeren Weg nehmen kann, wenn die minimale Entfernung mindestens 3 ist.

$W \quad 0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 4 \longrightarrow 5$

$F \quad 2 \longrightarrow 3 \longrightarrow 1 \longrightarrow 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 2$

H	i	0	1	2	3	4	5
	H_i	6	6+2	1	1+4	5+6	0 +4

Abbildung 6: Eine Erweiterung der Abb. 5. Die roten Zahlen stellen die addierten, generierten Ladungen dar, um die H_i vergrößert wurde.

Anschließend durchsuche ich das Array von allen Koordinaten, um die größte Koordinate zu finden. Ich speicher sie als $maxCoor$. Ich vergrößere diese Variable um 1. $maxCoor$ wird jetzt zur Länge und zur Breite der quadratischen, zu generierenden Matrix M. Ich tue das, um Schelifen bei den Batterien am Rand zu erlauben. Es macht die Aufgabe für eine Person vielleicht nicht schwieriger, aber es macht die Aufgabe schwieriger für den Rechner.
Am Ende wird die Länge/Breite der Matrix, die Koordinaten und die Ladung der Startbatterie, die Anzahl der restlichen Batterien und ihre Koordinaten und Ladungen im von BWINF vorgeschlagenen Format als eine Textdatei ausgegeben.

1.6.1 Laufzeit

n – die Anzahl der zu generierenden Batterien

- Generierung der Reihenfolge: $O(n)$
  In worst-case kann das augegebene Array maximal der Länge $2n$ sein, also: $O(2n)$.
- Generierung der Entfernungen und Koordinaten: $O(c * n)$ (worst-case)
  Das Anhängen der Entfernung im Array F erfolgt in $O(2n)$.
  Wir brauchen maximal $O(4 * currDist)$, um ein Paar von Koordinaten einer Etnfernung $currDist$ zu finden. $currDist$ ist am Anfang maximal 5. In average-case, wenn wir $currDist$ nicht vergrößern, beträgt die Laufzeit dann $O(1)$. In worst-case schätze ich die Anzahl der Vergrößerungen von $currDist$ nur als eine Konstante c ab, wobei $c \ll n$. Nach den durchgeführten Experimenten ist es sehr unwahrscheinlich, dass $currDist$ mehr als 1 Mal vergrößert werden muss.
- Generierung der Ladungen: $O(n)$
  Das Array mit der Reihenfolge wird iteriert ($O(2n)$) und das Array mit Ladungen wird iteriert, das maximal n lang sein kann ($O(n)$).

Nun addieren wir alles zusammen (worst-case):
$O(n) + O(c * n) + O(n) = O(c * n + 2n) \in O(n)$, wobei $c = const$.

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. Bei der Erstellung eines Objektes dieser Klasse kann man ein optionalen boolschen Parameter `slant` eingeben, der schräge Übergänge erlaubt. (s. Abschnitt 1.5.1)

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 Baterrien bekommen entsprechend die Indizes um 1 größer als ihre Vorgänger. Für die Bestimmung des Brettindex einer Batterie b bediene ich mich der folgender Formel:

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

Danach füge ich jeweilige Baterie 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 ihenen 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 $\texttt{boardDimension}^2$, 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 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.
Andernaflls 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` gekennzeichent.
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 betteriefreies 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 betteriefreies Feld befindet und dann überprüft, ob es noch ein batteriefreis Feld neben diesem Feld gibt.
Diese Funkiton 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 betteriefreie 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 interierten 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 `currChanges` 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 erriechbaren Batterien findet. Diese Liste speichern wir unter `neighbors`. Nun prüfen wird, ob diese Liste leer ist. Wenn ja, haben wir folgende Fallunterscheidungen.
Wenn `charge` gleich 0 ist, prüfen wir im Array `status`, ab alle Batterien entladen sind. Wenn ja, wird `foundPath` als `result` gespeichert und ein boolscher 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 wir geprüft, ob alle Ladungen im Arrax `status` 0 betragen. Wenn ja, wird `foundPath` als `result` gespeichert und ein boolscher 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 Arrax `status` 0 betragen und ob `charge` gleich 0 ist. Wenn ja, wird `foundPath` als `result` gespeichert und ein boolscher Wert 1 ausgegeben. Andernfalls wird 0 ausgegeben.
In allen anderen Fällen wird nun ein Array `allNeighbnors` gebildet. In das Array füge ich alle Eingabeindizes und alle minimale Entfernungen der Batterien, die erreichbar sind, unabhägig davon, ob sie entladen sind oder nicht. Ich sortiere dieses Array absteigend nach den minimlaen 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. Andern falls, 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 Arrax `status` 0 betragen und ob `charge` gleich 0 ist. Wenn ja, wird `foundPath` als `result` gespeichert und ein boolscher 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 `nieghDistances`, 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 bilden 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` ale 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 Deklarierung 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`

Diese Klassen beinhaltet einen Wert `batNum`, der die Anzahl der zu generierenden Batterien bestimmt. Der Wert `boardDimension` entspricht der Länge einer Seite der zu generierenden quadratischen Matrix. Ein Objekt der Klasse `Battery` namens `start` entspricht der Startbatterie. Anschließend befindet sich in dieser Klasse noch eine Menge von Objekten von `Battery`, die `batteries` genannt wird, die alle anderen Batterien (also nicht die Startbatterie) speichert. Für den Zufallszahlgenerator bediene ich mich des `std::random_device`

Die Methode `generateOrder()` generiert die Reihenfolge, in der die Batterien besucht werden. Ich bilde ein Array `num`, ein Array `generated` und einen Stapelspeicher `help`. Das erste Array enthälz die generierte Reihenfolge. Wir fügen 0 in es ein, weil die Reihenfolge stets mit der Startbatterie beginnt.
Mit Hilfe von `std::uniform_real_distribution` generiere ich `batNum -1` rationale Zahlen im Bereich $[0, 1)$. Ich runde sie dann auf Einer und füge in das Array `generated` ein. Das sind meine generierten Situationen: 0 ist Hinzufügen und 1 ist Einfügen.
Ich iteriere durch die Zahlen vom 1 bis `batNum` (ausschließlich). Jede iterierte Zahl nenne ich i. Wenn es sich an der Stelle $i - 1$ im Array `generated` 0 befindet, wird i am Ende des Arrays `num` angehängt. Außerdem wird das erste Element in `help` auch am Ende des Arrays `num` angehängt und danach wird dieses Element ausgekellert. Diese Operation erfolgt, so lange der Stapelspeicher nicht leer ist.
Wenn 1 an der Stelle $i - 1$ im Array `generated` steht, wird i am Ende von `num` angehängt. Dazu wird auch i in den Stapelspeicher eingekellert.
Anschließend erfolgt eine Ausgabe vom Array `num`.

In der Methode `generateDistances()` werden die minimalen Entfernungen zwischen Batterien und ihre Koordinaten generiert. Es wird ein Array `v` mit der Reihenfolge als Parameter genommen. Es werden folgende Arrays gebildet: `num` fürs Speichern der minimalen Entfernungen, `vis`, das boolsche Werte enthält, fürs Speichern die Batterien, für die koordinaten bestimmt wurden; `coor`, das Paar von ganzzahligen Koordinaten speichert. Außerdem bilde ich eine Menge `used`, die alle bereits benutzten Koordinaten speichert. Anschließend bilde ich einen Stapelspeicher `help`, der Paare von ganzen Zahlen, dem Index und der Entfernung, speichert.
An der Stelle 0 im Array `coor` speichere ich ein Paar von (0,0). Das sind die Koordinaten der Startbatterie. Nun erfolgt eine Iteration vom Array `v` bis zum vorletzten Element. Den Iterator nenne ich i. Als `index` speichere ich den Wert $\mathtt{v}_i$ und als `nextIndex` den Wert $\mathtt{v}_{i+1}$. Ich generiere eine ganze Zahl `dist` aus dem Bereich $[1, 5]$ anhand `std::uniform_int_distribution`. Ich kopiere `dist` zu `currDist`. Nun sehe ich das erste Element in `help` nach und überprüfe ob es gleich `nextIndex` ist. Wenn ja, nimmt `currDist` den Wert von diesem ersten ersten Element aus `help`.
Nun erfolgt eine Überprüfung anhand des Arrays `vis`, ob `nextIndex` bereits bearbeitet wurde. Wenn nicht, generieren wir neue Koordinaten für diese Batterie. Ich speichere die Koordinaten des Vorgängers von `nextIndex`, also `index`, als `prevPoint`. Gleichzeitig bilde ich zwei neue ganzzahligen Variablen `x`, `y` und eine boolesche Variable `all`.
Nun beginnt eine `do while`–Schleife, die läuft, so lange `all true` beträgt. In dieser Schleife bilde ich ein Array `free` mit booleschen Werten der Länge 4*`currDist`. Danach beginnt noch eine `do while`–Schleife. In dieser inneren Schleife wird eine ganze Zahl `coorID` aus dem Bereich [0, `4*currDist`) anhand `std::uniform_int_distribution` generiert. Anhand der Gleichung 1 werden die Variablen `x` und `y` bestimmt. Zu den Variablen `x` und `y` werden die Koordinaten von `prevPoint` entsprechend addiert. Danach wird `coorID` im Array `free` mit `true` gekennzeichnet. Dann bekommt `all` den Wert `true`. Es wird geprüft, ob an allen Stellen des Arrays `free` 1 steht. Wenn nicht, wird `all` zu `false`. Die innere `do while`–Schleife wird weiteriteriert, wenn das gefundene Paar von Koordinaten `x` und `y` sich bereits in der Menge `used` befindet und ob `all` gleich 0 ist. Die Suchoperation wird in der Laufzeitbetrachtung

vernachlässigt. Wenn nicht, bekommt dist den Wert currDist. currDist wird um 1 vergrößert. Hier endet die äußere do while–Schleife und wenn all false wird die Schleife abgebrochen.
An der Stelle nextIndex im Array coor wird das Par von neu gefundenen Koordinaten gespeichert. Diese Paar wird auch in die Menge used eingefügt. nextIndex wird im Array vis mit 1 gekennzeichnet.
Danach wird geprüft, ob der Index des ersten Elements aus help gleich nextIndex ist. Wenn ja, wird am Ende des Arrays num die Entfernung dieses ersten Elements angehängt und aus dem Stapelspeicher ausgekellert. Andernfalls, wird am Ende des Arrays num der Wert dist angehängt und ein Paar aus index und dist wird in help eingekellert. Hier endet die Iteration von v.
Anschließend wird die kleinste Koordinate von allen gefunden. Sie wird als minimal gespeichert. Dieser Wert wird mal -1 multipliziert und 2 wird dazu addiert. Nun vegrößern wir alle Koordinaten im Array coor um den Wert minimal.
Anschließend werden das Array num und das Array coor ausgegeben.

Die Methode generateCharges() generiert Ladungen für jede Batterie. Sie nimmt als Parameter ein Array v mit der Reihenfolge und ein Array dist mit den minimalen Entfernungen. Es werden ein Array ch und ein Array prev, beide der Länge batNum.
Falls das Array dist nicht leer ist, wird das Array v iteriert. Den Iterator nenne ich i. Als currDist speichere ich den Wert im Array dist an der Stelle i. Wenn der Wert an der Stelle $\mathtt{v}_i$ im Array ch gleich 0 ist, stelle ich an dieser Stelle den Wert currDist und wenn der Wert $\mathtt{v}_i$ nicht 0 ist, wird der Wert $\mathtt{v}_{i-1}$ an der Stelle $\mathtt{v}_i$ im Array prev gespeichert. Anderfalls, wenn der Wert an der Stelle $\mathtt{v}_i$ im Array ch nicht gleich 0 ist, addiere ich zu dem Wert an der Stelle $\mathtt{prev}_{\mathtt{v}_i}$ im Array ch den Wert currDist.
Danach wird das Array ch iteriert. Den Iterator nenne ich i.
Wen der Wert an der Stelle i im Array ch gleich 0 ist, wird ein ganzzahliger Wert multip im Bereich $[1, 3]$ anhand std::uniform_int_distribution generiert. Zu dem Wert an der Stelle i im Array ch addiere ich den 2. Vielfachen von multip. Andernfalls, der Wert an der Stelle i im Array ch nicht gleich 0 ist, tue ich das Gleiche, aber diesmal generiere ich einen Wert multip im Bereich $[0, 2]$.
Anschließend wird das Array ch ausgegeben.

Die Methode prepareOutput() bereitet die Textdatei auf die Ausgabe vor. Hier werden ein Array ch mit Ladungen und ein Array coor mit Koordinaten als Parameter genommen.
Ich iteriere durch die Liste von Koordinaten und suche nach der größten Koordinate von allen. Ich speichere sie als maximal. Ich vergrößere maximal um 1. Danach speichere ich diesen Wert als boardDimension, also die Länge der generierten Matrix.
Danach speichere ich unter start die Koordinaten und die Ladung der Startbatterie. Danach forlgt eine Iteration vom Array coor vom Index 1. Bei jedem iterierten Paar von Koordinaten speichere ich auch die dazugehörige Ladung als ein Objekt der Klasse Battery und füge es in die Menge batteries ein.

In der Methode save() werden die Informationen in start, boardDimension und batteries im von BWINF vorgeschlagenen Format gespeichert. Die Textdatei wird normalerweise in dem Order ../output/ unter stromrallyeNUM.txt, wobei NUM einer eingegeben Nummer entspricht, gespeichert.

Bei der Erstellung eines Objektes dieser Klasse muss man einen ganzzahligen Wert num eingeben. Dieser Wert muss mindestens 1 betragen. Er wird als batNum gespeichert. Es werden folgende Arrays gebildet: order, distances, coordinates und charges. Man lässt die Funktion generateOrder laufen und das Ergebnis wird als order gespeichert. Danach benutzt man dieses Array als Parameter in der Funktion generateDistances. Das erste ausgegebene Ergbenis dieser Funktion wird als distances und das zweite als coordinates gespeichert. Danach erfolgt ein Lauf der Funktion generateCharges mit order und coordinates als Parameter. Das ausgegebene Ergbenis speichere ich als charges. Anschließend lässt man die Funktion prepareOutput mit charges und coordinates laufen.

3 Beispiele

Die untenstehenden Zahlen, die die Folgen der Schitten 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 $a(d_a) \to b(d_b)$ bedeutet, dass man von der Batterie mit dem Eingabeindex a zur Betterie 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

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

3.2 Beispiel 1 (BWINF)

Textdatei: `stromrallye1.txt`

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

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

3.3 Beispiel 2 (BWINF)

Textdatei: `stromrallye2.txt`

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

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

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

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 Beipsiels 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 Beipsiels 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

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

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

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

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

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

3.11 Beispiel 3 (erweitert)

Textdatei: `stromrallye3.txt`
Besonderheit: schräge Übergänge sind erlaubt

Die Spielsituation ist lösbar.

Zeit: 0 min 0 s

$0 \to 2(9) \to 3(10) \to 1(2) \to 1(2) \to 1(2) \to 1(2) \to -1(1)$

4 Quellcode

stromrallye.m