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 1.2.1 präsentiert ein Beispiel, in dem diese besondere Unterscheidung wichtig ist.

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.

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: $b \to i \to j \to i \to b$. Auch hier können wir die in der Beobachtung 3 bemerkte Ordnungsgemäßheit anwenden, wenn wir l statt $A_{i,j}$ nehmen. (s. Abb. 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 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.

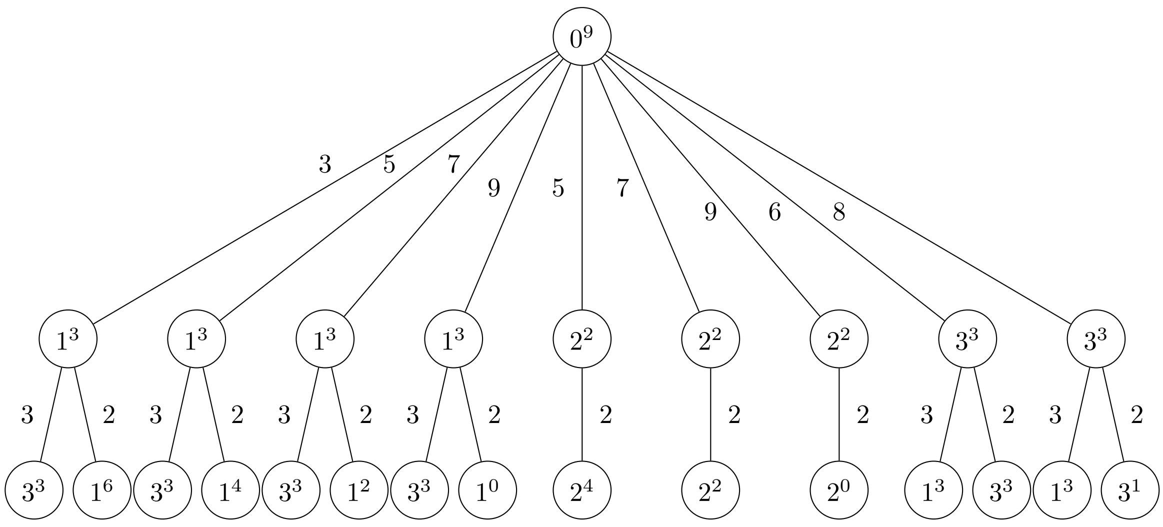

Abbildung 3: Ein abgebildeter Baum der 0., 1. und 2. Niveaus des Beispiels 3.1 (das Beispiel aus der Aufgabenstellung). Die Nummern in Knoten entsprechen den Eingabeindizes der Batterien. Die Potenzen stellen die Ladung dar, die wir auf einem Feld finden, wenn wir eine Batterie erreichen.

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.

Die erste aktuelle Ladung ist 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.
Wir lassen diese Funktion mit folgenden Parametern laufen: 0 als Eingabeindex, die aktuelle Ladung der Batterie s,

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 Esrstellung der Menge E: $O(l^2)$.
- Breitensuche: $O(b * l)$
  Die Laufzeit vom Breitensuchealgorithmus ist $O(V + E)$, $V = l$ 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 Bestimmu läuft in $O(1)$, also für alle Batterien: $O(b)$.

1.5 Generierung von Spielsituationen

1.5.1 Idee

1.5.2 Laufzeit

1.6 Erweiterungen

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 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 `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.
Andernafls 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`

2.3 Die Klasse `Generator`

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.
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) → 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

$0 \to -1(20)$

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

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

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

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

4 Quellcode

stromrallye.m