Aufgabe 1: Stromralley

Teilnahme-Id: 9693

Bearbeiter/-in dieser Aufgabe:
Nick Djerfi

20. April 2020

Inhaltsverzeichnis

1 Lösungsidee

Um Spielsituationen zu lösen wird ein Branch-and-Bound Bruteforce-Algorithmus angewendet, der mithilfe einer Heuristik und einer Bound-Bedingung in vielen Fällen schnell zu einer Lösung kommt. Eigene Spielsituationen werden generiert, indem zuerst, basierend auf den gewünschten Schwierigkeitsgrad, die Eigenschaften der Spielsituation festgelegt werden, und daraufhin die Spielsituation vom gelösten Zustand rückwärts in einen ungelösten Zustand "gelöst wird".

1.1 Branch-and-Bound Algorithmus

Der Algorithmus geht vom Startzustand alle Pfade ab, bis eine Lösung gefunden wurde. Dabei werden die einzelnen Pfade mittels einer Heuristik sortiert, damit Pfade, die zur Lösung führen können, zuerst bearbeitet werden. Es gibt eine Bound-Kondition, die frühzeitig erkennt, ob ein Pfad überhaupt noch zur Lösung führen kann. So kommt der Algorithmus entweder schnell zu einer Lösung, oder kann schnell erkennen, dass die Spielsituation unlösbar ist.

1.2 Branch-Heuristik

Jedem Pfad wird eine Punktzahl zugeordnet, an dieser der Algorithmus die Pfade sortiert. Diese Punktzahl setzt sich aus der Summe der einzelnen Ladungen zusammen. So wird die Ladungssumme des Pfades von der Gesamtladung der Ausgangssituation abgezogen. Weiterhin wird dieser Wert so skaliert, dass auf vollen Spielfeldern mit einer hohen Batteriedichte diese Punktzahl niedriger wird. Von dieser Punktzahl wird ein zweiter Faktor abgezogen. Dieser Faktor besteht aus der Abweichung des Pfades von einem "logischen" Pfad. Dieser logische Pfad ist ein Pfad, der einmal durch alle Felder geht und, wenn möglich, zur Startposition zurückkehrt. An jedem Schritt wird dessen Abweichung vom logischen Pfad berechnet. Diese Abweichung wird dann vom skalierten Ladungswert abgezogen, sodass bei volleren Spielfeldern die Abweichung vom logischen Pfad stärker ins Gewicht fällt. Bei den Beispieldateien `stromralley1.txt` und `stromralley2.txt` wird ein logischer Pfad bevorzugt, während bei `stromralley5.txt` ein schneller Pfad, der schnell alle Batterien entlädt, bevorzugt wird.

1.2.1 Hamiltonkreise und Pfade

Der logische Pfad, der durch alle Felder geht ist ein Hamiltonpfad. Wenn dieser Pfad einen Kreislauf bildet nennt man ihn einen Hamiltonkreis. Solche Hamiltonpfade lassen sich relativ leicht auf quadratischen Graphen generieren. Sehr einfach ist es bei geraden Seitenlängen, also bei 10x10 Spielfeldern zum Beispiel, da der Pfad sich dort einfach von oben nach unten schlängeln muss.

Schwieriger ist es bei ungeraden Seitenlängen, da dort ein Hamiltonkreis unmöglich ist. Damit der Pfad möglichst logisch bleibt, muss das Ende des Pfades möglichst nah am Anfang sein. Dazu geht der Pfad zuerst zu einer der Ecken. Darauf schlängelt sich der Pfad ebenfalls hin und her, aber diesmal nur in den vier Quadranten um die Startposition herum. Wenn der Pfad über eine Kante tritt, wird die Richtung in die der Pfad geht um 90° gedreht. Dieser Algorithmus findet in fasst allen Fällen direkt den kompletten Hamiltonpfad. Nur in wenigen Fällen lässt der Algorithmus ein Feld frei. Dieses befindet sich aber immer an der Startposition und kann deshalb einfach hinzugefügt werden. (siehe Abb. 1)

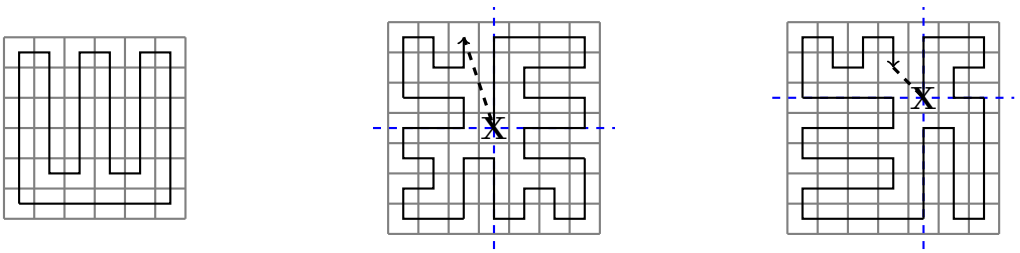


Abbildung 1: Hamiltonkreise

In der folgenden Abbildung 2 sind einige Beispiele für die Abweichung vom Hamiltonpfad dargestellt.

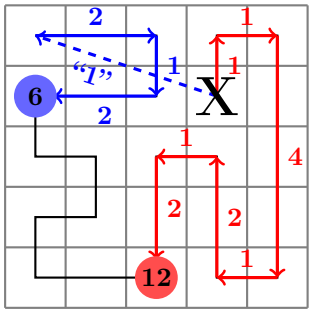
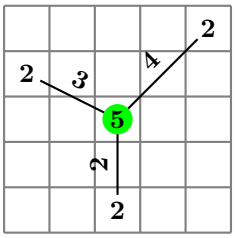


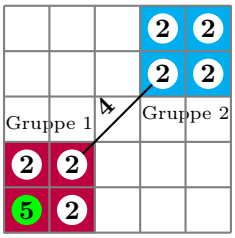
Abbildung 2: Abweichung vom Hamiltonpfad

1.3 Bound-Bedingung

Um unlösbare Spielsituationen frühzeitig zu erkennen, wird bei jedem Pfad überprüft, ob die Batterien in mehrere Gruppen unterteilt werden müssen. Eine Gruppe ist eine Ansammlung an Batterien, in denen jede Batterie möglicherweise jede andere Batterie erreichen kann. Kann der Algorithmus nicht alle Batterien zusammen mit dem Roboter in eine Gruppe fassen, ist die Spielsituation nicht mehr lösbar, da es für den Roboter dann keinen Weg zur anderen Gruppe gibt. Natürlich werden hier leere Batterien ignoriert, beziehungsweise als “Wände” gesehen.



(a) Eine große Gruppe



(b) Zwei kleine Gruppen

Abbildung 3: Gruppierung von Batterien

An sich führt dies aber zu falschen Ergebnissen, da man beachten muss, wenn Batterien mit höherer Ladung von anderen Batterien eingeschlossen sind. In Abbildung 4 würde der Algorithmus wie er gerade ist die Spielsituation als unlösbar einstufen, da die Batterie links von keiner der Batterien rechts erreichbar ist, weil der Roboter in der Mitte von diesen umschlossen ist und im Moment die Batterie links nicht direkt erreichen kann. Deshalb muss beim Gruppieren die maximale Ladung der einzelnen Gruppen zur Bestimmung, ob diese untereinander erreichbar sind, genommen werden. In der Abbildung ist die Maximalladung der Gruppe 9 und somit die Batterie links mit einer Distanz von 3 erreichbar.

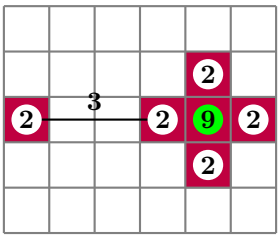


Abbildung 4: Edge-case

1.4 Generieren von Spielsituationen

Damit angemessene Spielsituationen erstellt werden können, muss zuerst definiert werden, was eine Spielsituation schwer macht. Aus diesem Schwierigkeitsgrad und der Spielfeldgröße lassen sich dann Spielfeldeigenschaften, wie die Anordnung der Batterien und die Länge des Lösungsweges generieren. Zuletzt wird, um die einzelnen Ladungen zu berechnen, die Spielsituation “rückwärts” gelöst, indem von der Zielkonfiguration schrittweise ein zufälliger Pfad zur Ausgangssituation generiert wird.

1.4.1 Berechnung des Schwierigkeitsgrades

Der Schwierigkeitsgrad einer Spielsituation setzt sich aus der Batteriedichte, der Länge des Lösungsweges und der Spielfeldgröße zusammen. Der Bereich des Schwierigkeitsgrades geht von 0 bis 100 für ein 20x20 großes Spielfelder.

Die Batteriedichte gibt an, wie dicht das Spielfeld mit Batterien voll ist. Eine Dichte von 1 bedeutet, dass alle Felder mit Batterien voll sind, wie in den den Beispieldateien `stromralley1.txt` und `stromralley2.txt`. Eine Dichte von 0 dahingegen bedeutet, dass das Spielfeld keine einzige Batterie hat, wie in Beispieldatei `stromralley4.txt`. Generell sind Spielsituationen mit hoher Batteriedichte schwerer als mit niedriger Batteriedichte. Gefühlt werden Spielsituationen mit hoher Batteriedichte nicht viel schwerer wird, wenn eine Batterie hinzugefügt wird, während Spielsituationen mit niedriger Batteriedichte um einiges schwerer werden, wenn die Batterieanzahl erhöht wird. Somit lässt sich der Schwierigkeitsgrad durch ein begrenztes Wachstum darstellen, wobei die Schranke einen Schwierigkeitsgrad von 40 darstellt. (siehe Abb. 5)

$$D(d) = 40 - 40 * e^{-5*d}$$

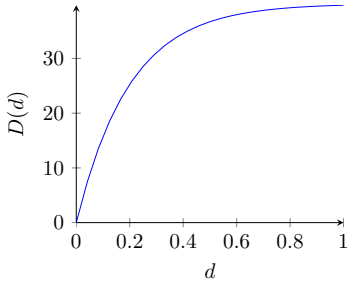


Abbildung 5: Schwierigkeitsgrad in Relation zur Batteriedichte

Die Länge des Lösungsweges gibt an, wie viele Batterien besucht werden müssen um die Spielsituation zu lösen. Eine Länge von 1 bedeutet, dass alle Batterien im Schnitt einmal besucht werden müssen, wohingegen eine Länge von 2 bedeutet, dass alle Batterien im Schnitt zweimal besucht werden müssen. Da mit steigender Länge die Möglichkeiten exponentiell steigen, lässt sich der Schwierigkeitsgrad durch eine Exponentialfunktion darstellen. Bei einer Lösungslänge < 1 nimmt die Anzahl an Möglichkeiten rasch ab, da dann immer mehr Batterien eine Ladung von 0 haben. Diese Batterien wirken wie Wände für den Roboter und grenzen dessen Möglichkeiten weiter ein. Deshalb wird ein weiterer Term angehängt, damit bei einer Lösungslänge < 1 die Funktion schnell abfällt. (siehe Abb. 6)

$$L(l) = (3^l - 1) * \frac{1}{l^3 + 2}$$

$$L(l) = \frac{3^l - 1}{l^3 + 2}$$

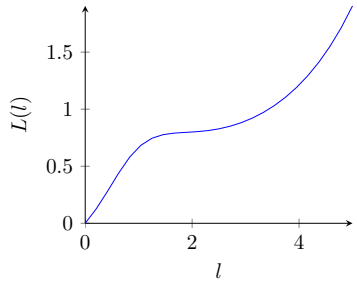


Abbildung 6: Schwierigkeitsgrad in Relation zur Lösungslänge

Einen Schwierigkeitsgrad erhält man aus der Kombination beider Funktionen. Dazu werden beide Funktionen miteinander multipliziert. (siehe Abb. 7)

$$DI(d,l) = D(d) * L(l)$$
$$DI(d,l) = (40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right)$$

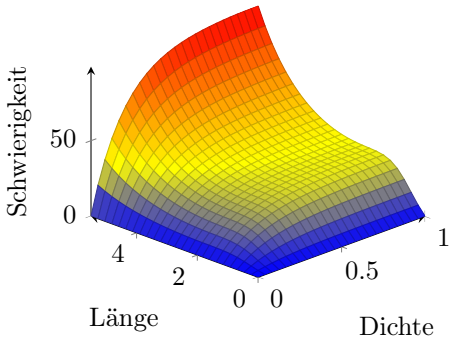


Abbildung 7: Schwierigkeitsgrad

Der Schwierigkeitsgrad nimmt quadratisch mit der Spielfeldgröße zu. Deshalb wird die Spielfeldgröße, zusammen mit der Batteriedichte und Lösungslänge, mit denn anderen beiden Funktionen addiert. (siehe Abb. 8)

$$f(n) = \frac{n^2}{100}$$

$$DI(d,l) = (40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * d * l$$

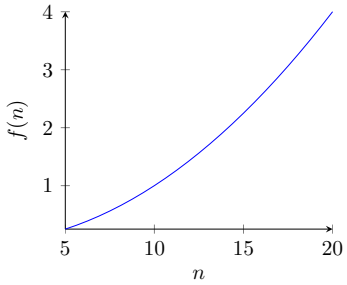


Abbildung 8: Schwierigkeitsgrad in Relation zur Spielfeldgröße

1.4.2 Herleitung der Spielfeldeigenschaften

Aus der Schwierigkeit und der Spielfeldgröße lässt sich nur schwer genau berechnen, welche Werte für die Batteriedichte und Lösungslänge möglich sind. Deshalb wird hier Gradient-Descent benutzt, um vom Mittelpunkt des Graphen bei $d = 0,5$ und $l = 2,5$ langsam zum gewünschten Schwierigkeitsgrad zu wandern. Für Gradient-Descent benötigt man die Ableitungsfunktionen in Relation zu den Unbekannten. (siehe Abb. 9)

$$
\begin{aligned}
dD(d,l) &= \frac{\partial}{\partial d}\Big(DI(d,l)\Big) \\
&= \frac{\partial}{\partial d}\left((40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * d * l\right) \\
&= \frac{\partial}{\partial d}\left((40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right)\right) + \frac{\partial}{\partial d}\left(\frac{n^2}{100} * d * l\right) \\
&= \frac{\partial}{\partial d}\left(40 - 40 * e^{-5*d}\right) * \left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * l \\
&= 200 * e^{-5*d} * \left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * l
\end{aligned}
$$

$$
\begin{aligned}
dL(d,l) &= \frac{\partial}{\partial l}\Big(DI(d,l)\Big) \\
&= \frac{\partial}{\partial l}\left((40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * d * l\right) \\
&= \frac{\partial}{\partial l}\left((40 - 40 * e^{-5*d}) * \left(\frac{3^l - 1}{l^3 + 2}\right)\right) + \frac{\partial}{\partial l}\left(\frac{n^2}{100} * d * l\right) \\
&= (40 - 40 * e^{-5*d}) * \frac{\partial}{\partial l}\left(\frac{3^l - 1}{l^3 + 2}\right) + \frac{n^2}{100} * d \\
&= (40 - 40 * e^{-5*d}) * \left(\frac{3^x \ln(3)}{x^3 + 2} - \frac{3(3^x - 1)x^2}{(x^3 + 2)^2}\right) + \frac{n^2}{100} * d
\end{aligned}
$$

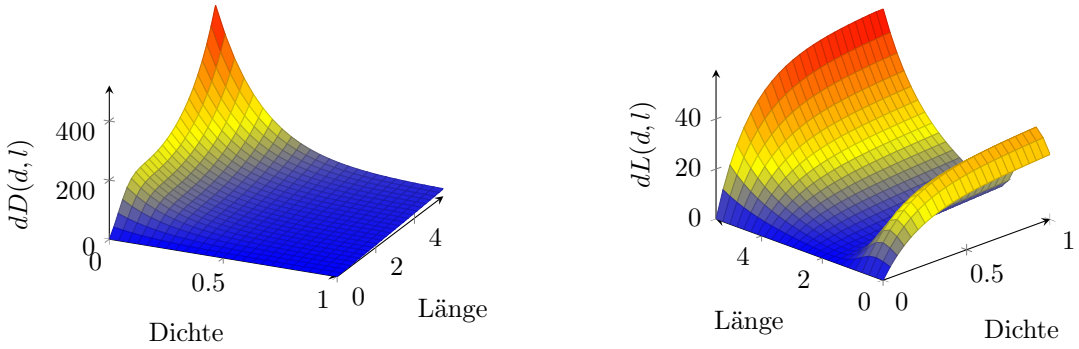


Abbildung 9: Ableitungsfunktionen

Nun kann man sich die Eigenschaft zu nutze machen, dass der Schwierigkeitsgrad mit steigenden Werten kontinuierlich steigt. So muss man nur vergleichen, ob sich der gewünschte Schwierigkeitsgrad überhalb oder unterhalb des derzeitigen Schwierigkeitsgrades befindet. Ist die gewünschte Schwierigkeit größer als die derzeitige, addiert man die Änderungsrate der jeweiligen Ableitungsfunktionen zu der Batteriedichte und der Lösungslänge. Andernfalls subtrahiert man die Änderungsrate von den Eingaben. Die Änderungsraten werden mit einer kleinen Schrittgröße von etwa $0,0001$ bis $0,001$ multipliziert, damit der gewünschte Schwierigkeitsgrad nicht übersprungen wird. (siehe Abb. 10)

1.4.3 Generieren der Spielsituation

Um aus diesen Informationen nun eine Spielsituation zu generieren, werden zuerst alle Batterien und die Start- und Zielposition zufällig auf dem Spielfeld verteilt. Daraufhin wird der Roboter auf die Zielposition mit einer Ladung von 0 gestellt. Dieser sucht sich zufällig eine der Batterien aus und fährt zu dieser. Dabei wird die Ladung des Roboters bei jedem Schritt um 1 erhöht. Kommt der Roboter auf ein Feld mit einer Batterie, wird wie im Spiel die Ladung des Roboters mit der Ladung der Batterie getauscht. Wenn der Weg zur Startposition länger wird als die gewünschte Lösungslänge, geht der Roboter den schnellsten

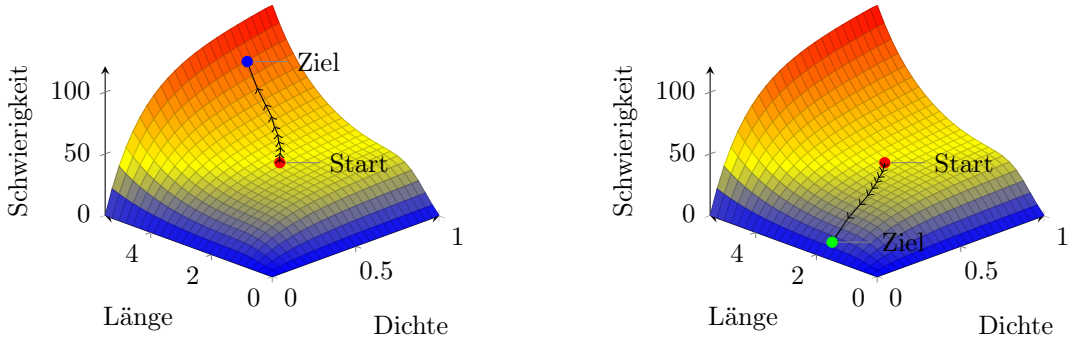


Abbildung 10: Gradient-Descent

Weg zur Startposition. Wenn der Roboter die Startposition erreicht hat, haben alle Batterien die richtige Ladung und der Roboter die richtige Position und Ladung. (siehe Abb. 11)

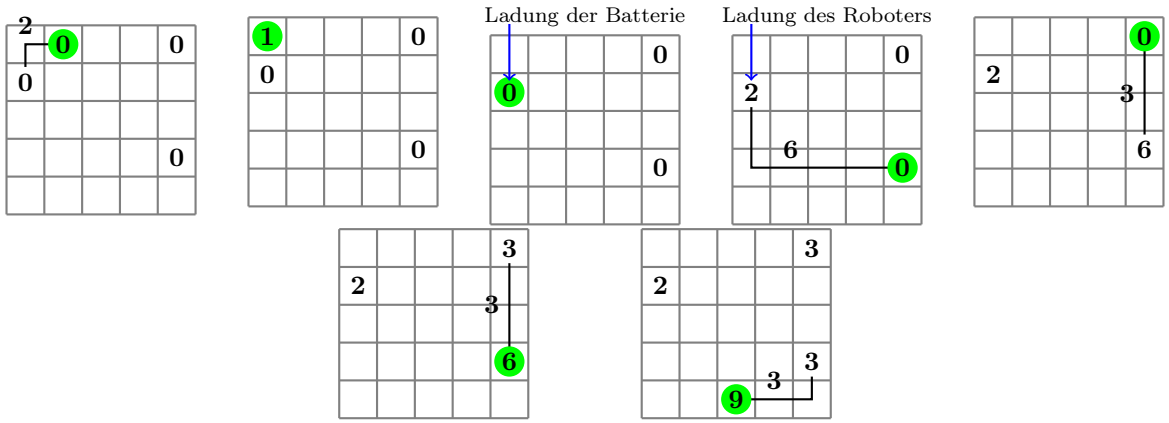


Abbildung 11: Generieren einer Spielsituation

2 Umsetzung

In der Umsetzung wurden viele Datentypen und Abstraktionen verwendet, um eine bessere Laufzeit zu erreichen.

2.1 Übersetzten von Punkten zu Indizes

Eine Abstraktion ist das Ersetzen der zweidimensionalen Punkte durch einfache Zahlen. Stellt man sich das Spielfeld als zweidimensionales Array vor kann man diese einfach in ein eindimensionales Format komprimieren. Dies hat den Effekt, dass sich die Adjazenzliste, Listen an besuchten Knoten und Ähnliches leichter und effizienter implementieren lassen. Da die Feldpositionen ab 1 anfangen muss 1 abgezogen werden.

```
uint32_t encode(point_t p, uint32_t size) {
  return (p.x - 1) + (p.y - 1) * size;
}

point_t decode(uint32_t n, uint32_t size) {
  return { (int)(n % size) + 1, (int)(n / size) + 1 };
}
```

2.2 Abstraktion des Spielfeldes

Das Spielfeld kann man als einen Graph sehen, bei dem die Batterien und der Roboter die Knoten sind und die Kanten die Pfade zwischen den Feldern speichern. Dies macht das Implementieren des Lösungs- und des Erstellungsalgorithmus einfacher, da der Roboter dann direkt von Batterie zu Batterie laufen

kann. Am Ende kann aus dieser Liste an Zügen und den Pfaden zwischen den Batterien der komplette Pfad rekonstrutiert werden. Weiterhin wird dieser Graph in einer Adjazenzliste gespeichert, was das Abrufen des Pfades zwischen zwei Batterien vereinfacht. (siehe Abb. 12)

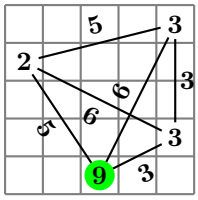


Abbildung 12: Abstraktion von `abbiegen0.txt`

2.2.1 Schleifenpfade

Nun kann es aber auch nötig sein, dieselbe Batterie nochmal zu besuchen, um diese zu leeren und mit der alten Ladung weiterzufahren. Um die linke Batterie in Abbildung 13 zu leeren muss der Roboter seine Ladung an dieser Batterie ablegen. Dadurch kann die Batterie geleert werden und die alte Ladung des Roboters wieder aufgenommen werden, mit der er zur Batterie rechts fahren kann.

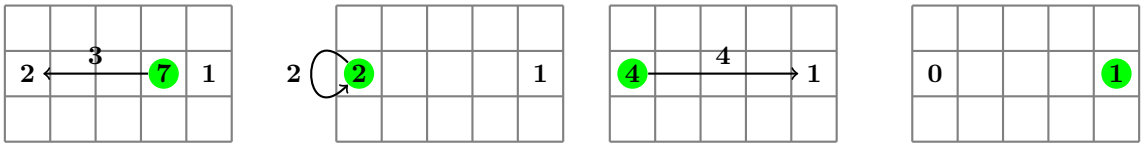
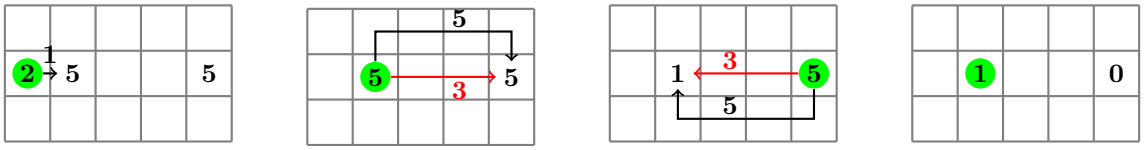


Abbildung 13: Beispiel von Schleifen

2.2.2 Längere Pfade

Letztlich muss, wenn der kürzeste Weg kürzer als vier Felder ist, auch ein verlängerbarer Pfad gespeichert werden. Ein verlängerbarer Pfad ist ein Pfad, auf dem der Roboter konstant hin und her fahren kann um eine größere Ladung zu entleeren. Damit dies möglich ist, muss ein Pfad mit mindestens zwei freie Felder haben, auf denen der Roboter hin und her fahren kann.



(a) Der Roboter hinterlässt eine Ladung von 1

(b) Der rote kürzeste Pfad macht es unlösbar

(c) Der rote kürzeste Pfad macht es wieder unlösbar

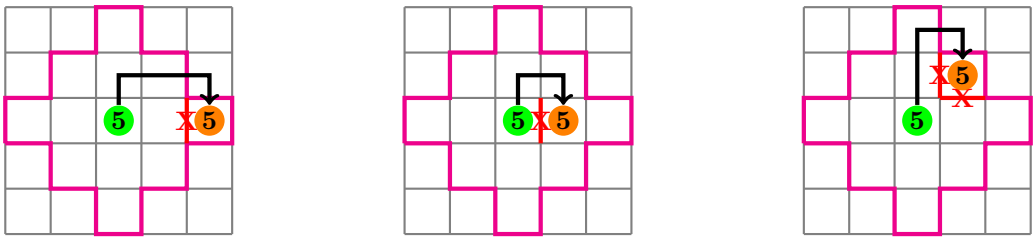
(d) Alle Batterien sind geleert

Abbildung 14: Beispiel von längeren Pfaden

2.2.3 Finden und speichern der Pfade

Diese Pfade werden mittels eines BFS-Algorithmus gesucht. Dabei kann angegeben werden, ob der Pfad zwingend verlängerbar sein muss. Ist der kürzeste Pfad schon verlängerbar kann dieser einfach dafür genommen werden, wenn nicht, wird der BFS nochmals mit diesem Parameter ausgeführt. Andere Batterien werden hier als Wände gesehen, da ein freier Pfad zwischen den beiden Batterien gefragt ist. Wird nach einem verlängerbaren Pfad gesucht wird zuerst geprüft, ob sich die Zielbatterie in einer Raute um die Startbatterie befindet. Anhand der Position der Zielbatterie in der Raute können entsprechend ein bis zwei Kanten im Graph blockiert werden, damit der BFS-Algorithmus diese nicht begeht. (siehe Abb. 15)

In einem Pfad Struct werden dann die jegliche Informationen und Pfade gespeichert. Am Ende erhält man eine Adjazenzliste, in der für jedes Batterienpaar ein solches Pfad Struct gespeichert ist.



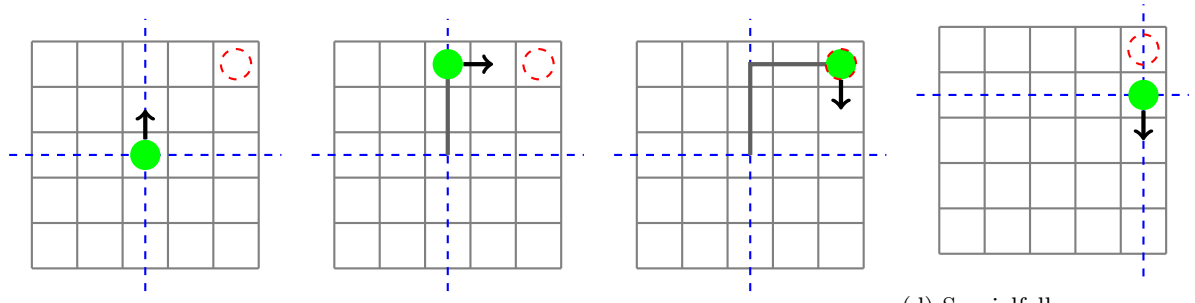
(a) Eine blockierte Kante (b) Eine blockierte Kante (c) Zwei blockierte Kanten

Abbildung 15: Rautenkonfigurationen

2.3 Hamiltonkreise

Hier ist der Algorithmus für gerade Spielfeldgrößen wesentlich einfacher als der für ungerade Größen. Bei geraden Größen wird einfach über einige If-Abfragen getestet, in welche Richtung gegangen werden muss. So wird, wenn der Roboter an der linken Wand ist, immer nach unten gegangen. Ist der Roboter an der unteren Wand wird immer nach rechts gegangen. Ansonsten wird basierend darauf, ob die x-Koordinate gerade oder ungerade ist, nach oben oder nach unten gegangen und an der Wand beziehungsweise kurz vor der Wand einmal nach links gegangen.

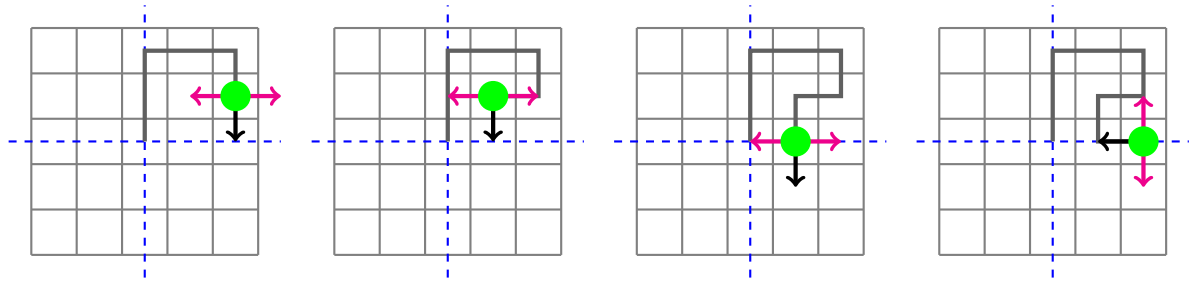
Anders ist es bei ungeraden Spielfeldgrößen. Dort wird eine Blickrichtung bestimmt, in der sich der Roboter schlängelt. Tritt der Roboter über die Ursprungsachsen wird die Blickrichtung um 90° rotiert. Am Anfang geht der Roboter aber zuerst, wenn möglich, in die naheste Ecke in einer Weise, dass er sich dannach um 90° drehen kann.



(a) Geht zur oberen Wand (b) Um 90° drehen (c) Geht zur Ecke (d) Spezialfall: Ecke unerreichbar

Abbildung 16: Zur nahesten Ecke gehen

Wenn die Ecke nun erreicht ist – oder auch nicht – schlängelt sich der Pfad solange hin und her, bis die blau gestrichelte Linie übertreten wird. Hierzu wird zusätzlich zur Blickrichtung noch die Orthogonale dazu gebildet. Der Roboter versucht solange die Orthogonale entlangzulaufen, bis kein Feld in dieser frei ist. Wenn kein Feld frei ist geht der Roboter einen Schritt in die Blickrichtung. Hat der Roboter die gestrichelte Linie übertreten und befindet sich an einer Wand kann die Blickrichtung wieder rotiert werden.



(a) In Blickrichtung gehen (b) Die Orthogonale entlang (c) In Blickrichtung gehen (d) Um 90° drehen

Abbildung 17: Hin und her schlängeln

Dies wird solange wiederholt, bis alle Felder durchloffen wurden, oder der Algorithmus aus dem Spiel-

feld austritt. Der Algorithmus geht aus dem Spielfeld raus, wenn noch Felder fehlen. Es kann nämlich vorkommen, dass ein einziges Feld fehlt. Dieses befindet sich immer neben der Startposition und kann einfach am Anfang des Pfades angefügt werden.

2.3.1 Ring Datenstruktur

Dieser Hamiltonkreis wird in einer Doubly-Linked List gespeichert. Dabei verweist die erste Position auf die letzte, sodass ein Ring gebildet wird. Der Algorithmus speichert einen Zeiger auf eine der Nodes im Ring. Von dieser Node aus kann man nun andere Nodes suchen und deren Adresse zurückgeben. So wird der Ring "rotiert" indem der Zeiger einfach auf diese neue Node gesetzt wird. Weiterhin kann man die Distanz zwischen zwei Punkten im Ring berechnen, indem man zählt, wie viele Schritte benötigt werden, um den anderen Punkt zu erreichen. Man kann das Vorzeichen dieser Distanz bestimmen, indem man einmal den Ring in die eine Richtung untersucht mit dem Zeiger auf die nächste Node. Dies ist dann die positive Distanz. Die negative Distanz erhält man, indem man den Ring in der anderen Richtung untersucht mit dem Zeiger auf die vorherige Node. Zurückgegeben wird die Distanz mit dem kleineren Betrag. Dies wird in der Punktzahlberechnung der einzelnen Pfade verwendet, indem es eine "Strafe" dafür gibt, wenn sich das Vorzeichen des Pfades im nächsten Schritt umdreht.

2.4 Lösungsalgorithmus

Der Lösungsalgorithmus sucht zuerst mit dem Bruteforce-Algorithmus einen Lösungsweg auf dem abstrahierten Graphen. Aus diesen Zügen wird dannach die Folge an Schritten rekonstrutiert und zurückgegeben.

2.4.1 Gruppenüberprüfung

Zur Gruppenüberprüfung wird die Liste an Batterien durchiteriert. Für jede Batterie wird überprüft, ob diese zu einer der vorhandenen Gruppen gehört. Gehört die Batterie zu mehreren Gruppen werden diese zu einer großen Gruppe zusammengefasst. Zuletzt wird die Batterie zu der Gruppe hinzugefügt. Wenn die Batterie zu keine der Gruppen gehört, wird eine neue angelegt. Jede Gruppe speichert zusätzlich die Batterie mit der höchsten Ladung, da diese Ladung zum Überprüfen verwendet wird. Am Ende wird zurückgeben, ob es mehr als eine große Gruppe gibt, da so die Spielsituation unlösbar ist.

2.4.2 Bruteforce Algorithmus

Der Algorithmus verhält sich ähnlich wie ein Dijkstra Algorithmus, da ebenfalls in einer Prioritätswarteschlange die einzelnen Pfade abgearbeitet werden. Initialisiert wird diese Warteschlange mit der Startposition. Von der aus werden alle benachbarten Batterien in der Adjazenzliste durchiteriert. Weiterhin werden für jede benachbarte Batterie die möglichen Distanzen durchiteriert. Das heißt, einmal die kürzeste Distanz und, wenn vorhanden, alle verlängerten Distanzen von der Mindestdistanz bis hin zur Ladung des Roboters.

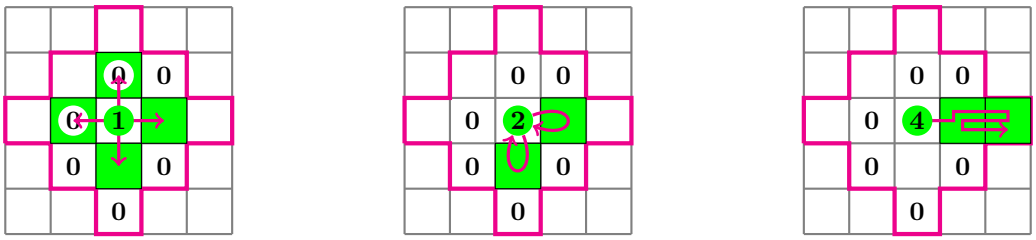
Mit dieser Distanz wird dann zuerst der derzeitige Pfad kopiert und ein Zug angefügt. Für diesen neuen Pfad wird, bevor dieser zur Warteschlange hinzugefügt wird, noch die neue Punktzahl berechnet.

So werden immer neue Pfade in die Warteschlange gelegt und immer der Pfad, mit der besten Punktzahl bearbeitet, bis die Zielsituation erreicht wurde. Wenn alle Batterien eines Pfades geleert wurden wird, wie wieder mithilfe der Raute ein Pfad zum Entleeren der Bordbatterie gesucht. Wenn die Bordbatterie nur eine Ladung von 1 hat wird eines der benachbarten Felder ausgewählt. Wenn die Bordbatterie eine Ladung von 2 hat muss dieses Feld frei sein, damit der Roboter darauf gehen kann und wieder zurück. Andernfalls muss es zwei aufeinanderfolgende freie Felder innerhalb der Raute geben, damit der Roboter darauf hin und her fahren kann. (siehe Abb. 18)

2.4.3 Worker Struct

Jeder der Pfade wird mit einem Worker Struct repräsentiert, dieses speichert die Zugfolge, die Punktzahl des Pfades und dessen Laufrichtung im Hamiltonkreis. Der Vergleichsoperator dieser Pfade vergleicht die Punktzahl, damit die Pfade in der Prioritätswarteschlange richtig sortiert werden. Zur Punktzahlberechnung wird zuerst die entladene Ladungsmenge berechnet und mit der Batteriedichte skaliert.

```
1 score = map.chargeSum - (state.chargeSum - this->path.back().distance);
  score *= map.area / state.batteries.size();
```



(a) Egal welches Feld (b) Ein freies Feld (c) Zwei freie Felder

Abbildung 18: Entladen der Bordbatterie

Dannach wird die Abweichung vom Hamiltonkreis berechnet und, je nach dem ob ein Richtungswechsel stattfand, doppelt oder an sich von der Punktzahl abgezogen.

```
// Hamiltonkreis zur alten Position rotieren
hamiltonCycle = hamiltonCycle->find(state.robot.position);

// Abweichung zur neuen Position berechnen
int signedDistance = hamiltonCycle->distance(this->path.back().position);
int sign = sgn(signedDistance);

// Fand ein Richtungswechsel statt?
if (sign != prevDirection) {
  // "Strafe" fuers umdrehen
  signedDistance *= 2;
  prevDirection = sign;
}

// Abweichung von der Punktzahl abziehen
score -= std::abs(signedDistance);
```

2.4.4 Rekonstruktion der Lösung

Zur Rekonstruktion des Lösungsweges werden nun alle Züge durchiteriert und eine Liste an Schritten erstellt. Dazu wird für jeden Zug in der Zugfolge das Pfad Struct abgerufen. Ist die Länge dessen kürzesten Pfades gleich wie die gegangene Distanz im Zug wird dieser Pfad zur Liste an Schritten hinzugefügt. Ist die Länge unterschiedlich wurde der verlängerbare Pfad im Zug gegangen. Dazu wird zuerst berechnet, um wie viel der verlängerbare Pfad verlängert wurde. Für diese Verlängerung werden immer wiederholt das erste und zweite Feld zur Schrittliste hinzugefügt. Wenn die Verlängerung abgearbeitet wurde wird der restliche verlängerbare Pfad hinzugefügt.

Basierend auf die Restladung des Roboters werden die letzten Züge ausgelassen, da diese die freien Felder um den Roboter herum speichern und somit eventuell keinen Eintrag in der Adjazenzliste haben. Für die Restladung werden diese Felder einfach am Ende angefügt. Wenn die Restladung größer als 2 ist müssen diese Felder solange wiederholt werden bis der Roboter entladen ist.

2.5 Generator-Algorithmus

Der Generator-Algorithmus berechnet zuerst aus den übergebenen Bedingungen die Spielfeldeigenschaften und konstruiert aus diesen dann eine Spielsituation. Die Bedingungen beinhalten die Spielfeldgröße, den Schwierigkeitsbereich und den möglichen Bereichen für die Batteriedichte und Lösungslänge.

2.5.1 Lösen der Bedingungen

Die Spielfeldeigenschaften, die zum gewünschten Schwierigkeitsgrad führen, werden iterativ über einen Gradient-Descent Algorithmus bestimmt. Hierzu werden die Eigenschaften mit den Mittelwerten der erlaubten Bereiche initialisiert. Dannach wird in jeder Iteration die Steigung der Funktion in Bezug auf die Variablen berechnet. Je nach dem, ob sich der Zielschwierigkeitgrad über oder unter dem derzeitigen befindet, wird der Kehrwert der Steigung multipliziert mit der Schrittgröße addiert oder subtrahiert. Durch den Kehrwert werden auf flachen Gebieten der Schwierigkeitsgradfunktion größere Schritte, als bei steileren Gebieten gemacht, um den gewünschten Wert schnell zu erreichen und nicht zu überspringen. Wenn durch den Schritt eine der beiden Variablen aus dem erlaubten Bereich geht, wird der Schritt

für diese Variable rückgängig gemacht. Dies hat den Effekt, dass sich der Punkt an der Grenze zum gewünschten Wert entlangbewegt. (siehe Abb. 19)

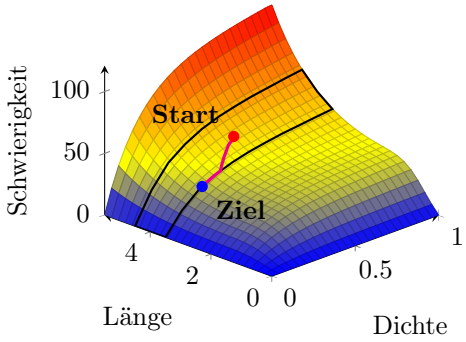
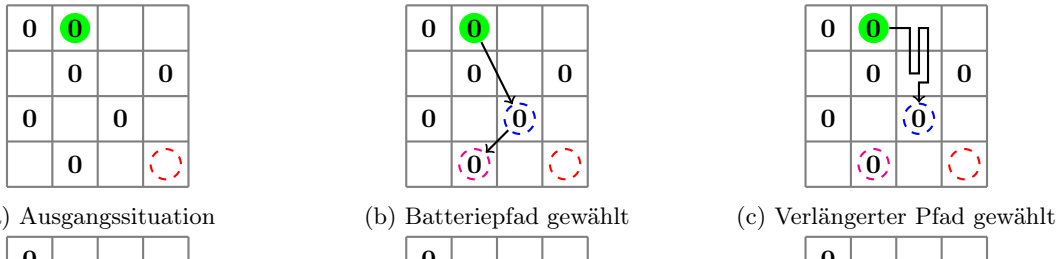


Abbildung 19: Gradient-Descent mit Grenzen

2.5.2 Konstruieren der Spielsituation

Mit diesen berechneten Eigenschaften kann man nun ein Spielfeld generieren. Dazu werden zu erst die Start- und Zielposition und die Batterien zufällig auf dem Spielfeld verteilt. Für diese Verteilung wird die abstrahierte Adjazenzliste generiert und gespeichert. Alle Ladungen sind zu Anfang mit 0 initialisiert und der Roboter befindet sich zu Anfang auf der Zielposition. (siehe 20a) Dannach wird, solange der Roboter sich nicht an der Startposition befindet, immer eine zufällige Batterie ausgewählt. Daraufhin wird ein Pfad zu dieser Batterie gesucht, der dann vom Roboter gegangen wird. Dazu wird ein BFS Algorithmus auf dem abstrahierten Graphen angewandt, der von Batterie zu Batterie geht um zur Zielbatterie zu gelangen. (siehe 20b) Für jede Batterie wird dann per Zufall entweder der kürzeste Pfad, oder wenn möglich der verlängerbare Pfad genommen. (siehe 20c u. 20e) Zur Ladung des Roboters wird die Länge des Pfades addiert, da von der Zielsituation rückwärts gegangen wird. Zuletzt tauscht der Roboter seine Ladung mit der Batterie. (20d u. 20f) Nach jeder Batterie wird die Lösungslänge um 1 verringert.



(a) Ausgangssituation (b) Batteriepfad gewählt (c) Verlängerter Pfad gewählt

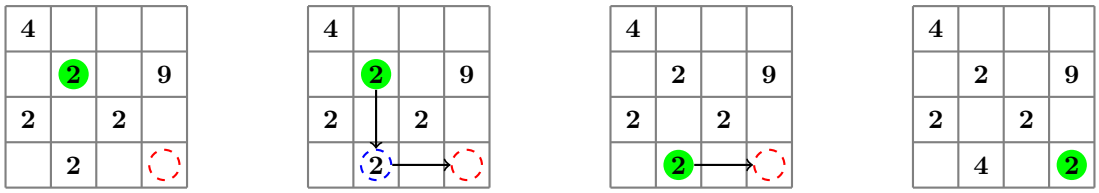
(d) Pfad gegangen (e) Kürzesten Pfad gewählt (f) Pfad gegangen

Abbildung 20: Rückwärts lösen

Ist die Lösungslänge kleiner als die Länge des Weges zur Startposition, geht der Roboter sofort diesen entlang. Damit die Batterien gleichmäßig geladen werden, wird für jede Batterie gezählt, wie oft diese besucht wurde. Bei der Wahl der nächsten Batterie wird dann die am wenigsten besuchte Batterie gewählt.

2.6 Laufzeitanalyse

Der Lösungsalgorithmus ist vergleichbar mit einem A*-Algorithmus. Er benutzt eine Heuristik um einen Pfad auf einem Graphen zu finden. Dennoch ist die Laufzeitkomplexität des Algorithmus höher als die des A*-Algorithmus, da die Dynamik des Graphen keine zulässige Heuristik erlaubt und mehrfaches Besuchen von Knoten erfordert. Also kann davon ausgegangen werden, dass die Komplexität größer ist als $O(b^d)$, wo b der Verzweigungsfaktor ist und d die Länge des Lösungsweges.



(a) Ausgangssituation (b) Pfad zur Startposition (c) Gehe zur Batterie (d) Gehe zum Start

Abbildung 21: Zur Startposition gehen

Die echte Laufzeit des Algorithmus kommt stark auf die Spielsituation an. In manchen Fällen terminiert der Algorithmus schon nach wenigen Millisekunden, in anderen kann es einige Minuten dauern. Generell steigt die Laufzeit exponentiell mit dem Schwierigkeitsgrad der Spielsituation an. Bei generierten Spielsituation mit einem Schwierigkeitsgrad von 0 bis etwa 35 terminiert der Algorithmus in unter einer Minute. Zwischen 35 und 45 schwankt die Laufzeit von Spielsituation zu Spielsituation. Bei höheren Schwierigkeitsgraden terminiert der Algorithmus erst nach vielen Minuten. In seltenen Fällen benötigt der Algorithmus selbts bei leichteren Spielsituationen mehrere Minuten. Dies geschieht eher bei großen Spielfeldern. Persönlich würde ich auch sagen, dass die Spielsituationen mit Schwierigkeitsgraden von über 40 für einen Menschen nahezu unmöglich werden. In der Tabelle 1 sieht man den drastischen Sprung ins Unberechenbare für den Algorithmus bei höheren Schwierigkeitsgraden.

Zeit t [s]		Schwierigkeitsgrad			
		10	20	30	50
Feldgröße	5x5	0,005	0,006	0,006	0,180
	10x10	0,006	0,008	>3min	>3min
	15x15	0,008	>3min	>3min	>3min
	20x20	0,019	>3min	>3min	>3min

Tabelle 1: Zeiten für Lösungsalgorithmus [Intel i7-6700 @ 3,40 GHz]

Der Generator-Algorithmus hingegen braucht für größere Spielfelder quadratisch länger, da dadurch die Batterieanzahl und Lösungslänge quadratisch ansteigt. Mit höherem Schwierigkeitsgrad hingegen steigt die Batterieanzahl und Lösungslänge nur etwa linear im Bereich von 0 bis 100. Deshalb kann man von einer Laufzeitkomplexität von $O(d * n^2)$ ausgehen, wo n die Spielfeldgröße und d der Schwierigkeitsgrad ist.

Zeit t [ms]		Schwierigkeitsgrad					
		10	20	30	50	75	100
Feldgröße	5x5	1	1	1	2	2	3
	10x10	2	3	6	10	21	23
	15x15	5	22	40	78	134	180
	20x20	19	101	224	452	1543	1904

Tabelle 2: Zeiten für Generator-Algorithmus [Intel i7-6700 @ 3,40 GHz]

2.7 Bedienung

Beide Algorithmen sind in einem Programm zusammengefasst. Dieses lässt sich über die Kommandozeile starten und gibt die Ergebnisse entweder auf der Konsole aus oder in einer Datei.

2.7.1 Eingabe

Zum Ausführen des Programm muss immer der Modus angegeben werden. Im Lösungsmodus ist die Datei die Eingabedatei deren Spielsituation gelöst werden soll. Im Generator-Modus ist die Datei die Ausgabedatei in der die generierte Spielsituation gespeichert werden soll.

```
$ ./stromralley <Modus> <Datei>
$ ./stromralley solve ../Beispiele/stromralley0.txt
$ ./stromralley generate meinBeispiel.txt
```

Für den Generator-Modus können die Eigenschaften und Begrenzungen als Parameter angegeben werden. Ohne Parameter ist die Standardgröße ein 10x10 Spielfeld, auf dem die Eigenschaften zufällig gewählt werden. Hierbei kommt zuerst die Größe des Spielfeldes, dann der minimale und maximale Schwierigkeitsgrad, die minimale und maximale Batteriedichte und die minimale und maximale Lösungslänge.

Das folgende Beispiel generiert eine Spielsituation mit einem 10x10 großen Spielfeld. Diese hat einen Schwierigkeitsgrad zwischen 15 und 30, eine Batteriedichte zwischen 0,2 und 0,5 und eine Lösungslänge von 0,5 bis 1,5.

```
$ ./stromralley generate meinBeispiel.txt 10   15 30   0.2 0.5   0.5 1.5
```

Diese Werte können auch explizit angegeben werden, wenn man zum Beispiel nur die Batteriedichte bestimmen möchte.

kurz	lang	Beschreibung
`-s`	`--size`	Spezifiziert die Spielfeldgröße
`-d`	`--min-difficulty`	Spezifiziert den minimalen Schwierigkeitsgrad
`-D`	`--max-difficulty`	Spezifiziert den maximalen Schwierigkeitsgrad
`-b`	`--min-density`	Spezifiziert die minimale Batteriedichte
`-B`	`--max-density`	Spezifiziert die maximale Batteriedichte
`-l`	`--min-length`	Spezifiziert die minimale Lösungslänge
`-L`	`--max-length`	Spezifiziert die maximale Lösungslänge

2.7.2 Ausgabe

Im Lösungsmodus wird am Ende der Lösungsweg ausgegeben.

```
$ ./stromralley solve ../Beispiele/stromralley0.txt
Lösungsweg:
(4|5) (5|5) (5|4) (5|3) (5|2) (5|1) (5|2) (5|3) (5|4) (4|4)
(3|4) (2|4) (1|4) (1|3) (1|2) (1|3) (1|2)
```

Im Generator-Modus werden zuerst die Eigenschaften der Spielsituation ausgegeben. Daraufhin wird die Spielsituation auf der Konsole ausgegeben. Zuletzt wird ausgegeben, ob das Schreiben der Ausgabedatei erfolgreich war.

```
$ ./stromralley generate meinBeispiel.txt
Eigenschaften der Spielsituation
--------------------------------
Schwierigkeitsgrad: 19.7235
Batteriedichte    : 0.135726
Lösungslänge      : 0.116054
Anzahl Batterien  : 14

Spielsitation:
--------------
10
3,6,5
14
8,1,17
1,2,8
4,2,3
9,2,5
10,2,1
5,3,6
5,4,11
2,5,4
7,5,5
4,6,0
7,6,0
10,8,0
8,9,0
2,10,0

Datei wurde erfolgreich erstellt: meinBeispiel.txt
```

3 Beispiele

Im folgenden werden zuerst die Ergebnisse des Programms bei allen Beispieldateien aufgelistet, daraufhin wird anhand des ersten Beispiels die Funktionsweise anschaulich erläutert.

3.1 Ergebnisse

Zu allen Beispieldateien findet der Algorithmus eine richtige Lösung.

```
$ ./stromralley solve ../Beispiele/stromralley0.txt
Lösungsweg:
(4|5) (5|5) (5|4) (5|3) (5|2) (5|1) (5|2) (5|3) (5|4) (4|4) (3|4) (2|4)
(1|4) (1|3) (1|2) (1|3) (1|2)

$ ./stromralley solve ../Beispiele/stromralley1.txt
Lösungsweg:
(2|1) (2|2) (2|3) (2|4) (2|5) (2|6) (2|7) (2|8) (2|9) (3|9) (3|8) (3|7)
(3|6) (3|5) (3|4) (3|3) (3|2) (3|1) (4|1) (4|2) (4|3) (4|4) (4|5) (4|6)
(4|7) (4|8) (4|9) (5|9) (5|8) (5|7) (5|6) (5|5) (5|4) (5|3) (5|2) (5|1)
(6|1) (6|2) (6|3) (6|4) (6|5) (6|6) (6|7) (6|8) (6|9) (7|9) (7|8) (7|7)
(7|6) (7|5) (7|4) (7|3) (7|2) (7|1) (8|1) (8|2) (8|3) (8|4) (8|5) (8|6)
(8|7) (8|8) (8|9) (9|9) (9|8) (9|7) (9|6) (9|5) (9|4) (9|3) (9|2) (9|1)
(10|1) (10|2) (10|3) (10|4) (10|5) (10|6) (10|7) (10|8) (10|9) (10|10)
(9|10) (8|10) (7|10) (6|10) (5|10) (4|10) (3|10) (2|10) (1|10) (1|9) (1|8)
(1|7) (1|6) (1|5) (1|4) (1|3) (1|2) (1|1)

$ ./stromralley solve ../Beispiele/stromralley2.txt
Lösungsweg:
(5|6) (4|6) (3|6) (2|6) (1|6) (1|5) (1|4) (1|3) (1|2) (1|1) (2|1) (2|2)
(2|3) (2|4) (2|5) (3|5) (3|4) (3|3) (3|2) (3|1) (4|1) (4|2) (4|3) (4|4)
(4|5) (5|5) (5|4) (5|3) (5|2) (5|1) (6|1) (6|2) (6|3) (6|4) (6|5) (7|5)
(7|4) (7|3) (7|2) (7|1) (8|1) (9|1) (10|1) (11|1) (11|2) (10|2) (9|2) (8|2)
(8|3) (9|3) (10|3) (11|3) (11|4) (10|4) (9|4) (8|4) (8|5) (9|5) (10|5)
(11|5) (11|6) (10|6) (9|6) (8|6) (7|6) (7|7) (8|7) (9|7) (10|7) (11|7)
(11|8) (11|9) (11|10) (11|11) (10|11) (10|10) (10|9) (10|8) (9|8) (9|9)
(9|10) (9|11) (8|11) (8|10) (8|9) (8|8) (7|8) (7|9) (7|10) (7|11) (6|11)
(6|10) (6|9) (6|8) (6|7) (5|7) (5|8) (5|9) (5|10) (5|11) (4|11) (3|11)
(2|11) (1|11) (1|10) (2|10) (3|10) (4|10) (4|9) (3|9) (2|9) (1|9) (1|8)
(2|8) (3|8) (4|8) (4|7) (3|7) (2|7) (1|7) (2|7) (3|7) (4|7) (4|8) (3|8)
(2|8) (1|8) (1|9) (2|9) (3|9) (4|9) (4|10) (3|10) (2|10) (1|10) (1|11)
(2|11) (3|11) (4|11) (5|11) (5|10) (5|9) (5|8) (5|7) (6|7) (6|8) (6|9)
(6|10) (6|11) (7|11) (7|10) (7|9) (7|8) (8|8) (8|9) (8|10) (8|11) (9|11)
(9|10) (9|9) (9|8) (10|8) (10|9) (10|10) (10|11) (11|11) (11|10) (11|9)
(11|8) (11|7) (10|7) (9|7) (8|7) (7|7) (7|6) (8|6) (9|6) (10|6) (11|6)
(11|5) (10|5) (9|5) (8|5) (8|4) (9|4) (10|4) (11|4) (11|3) (10|3) (9|3)
(8|3) (8|2) (9|2) (10|2) (11|2) (11|1) (10|1) (9|1) (8|1) (7|1) (7|2)
(7|3) (7|4) (7|5) (6|5) (6|4) (6|3) (6|2) (6|1) (5|1) (5|2) (5|3) (5|4)
(5|5) (4|5) (4|4) (4|3) (4|2) (4|1) (3|1) (3|2) (3|3) (3|4) (3|5) (2|5)
(2|4) (2|3) (2|2) (2|1) (1|1) (1|2) (1|3) (1|4) (1|5) (1|6) (1|7) (2|7)
(2|6) (3|6) (4|6) (5|6) (6|6)

$ ./stromralley solve ../Beispiele/stromralley3.txt
Die Spielsituation ist unlösbar!

$ ./stromralley solve ../Beispiele/stromralley4.txt
Lösungsweg:
(40|26) (40|27) (40|26) (40|27) (40|26) (40|27) (40|26) (40|27) (40|26) (40|27)
(40|26) (40|27) (40|26) (40|27) (40|26) (40|27) (40|26) (40|27) (40|26) (40|27)

$ ./stromralley solve ../Beispiele/stromralley5.txt
Lösungsweg:
(11|15) (12|15) (13|15) (14|15) (14|16) (14|17) (14|18) (15|18) (16|18) (17|18)
(18|18) (18|19) (18|20) (19|20) (19|19) (19|20) (19|19) (19|20) (19|19) (19|20)
(19|19) (19|18) (18|18) (18|17) (18|16) (18|15) (17|15) (16|15) (15|15) (14|15)
(14|14) (14|13) (14|12) (14|11) (14|10) (14|9) (14|10) (14|9) (13|9) (12|9)
(11|9) (10|9) (9|9) (8|9) (7|9) (6|9) (5|9) (4|9) (5|9) (5|8) (5|9) (5|8) (5|7)
(5|6) (5|5) (5|4) (5|3) (5|4) (4|4) (4|5) (4|6) (4|7) (4|8) (3|8) (2|8) (1|8)
(1|9) (1|10) (1|11) (1|12) (2|12) (2|13) (1|13) (1|14) (1|15) (1|16) (1|17)
(2|17) (3|17) (4|17) (4|18) (3|18) (2|18) (2|19) (3|19) (2|19) (1|19) (1|18) (1|19)
```

3.2 Beispiel an `stromralley0.txt`

Im folgenden wird die Funktionsweise des Lösungsalgorithmus an der Beispielsituation erläutert. Hier wird zum einen darauf eingegangen wie die Abstraktion des Spielfeldes aussieht und wie daraus eine Lösung gebildet wird.

3.2.1 Spielfeldabstraktion

In der folgenden Abbildung 22 ist die Abstraktion des Spielfeldes dargestellt. Dabei bedeutet die erste Zahl über den Kanten die Länge des kürzesten Pfades und die zweite Zahl die des verlängerbaren Pfades.

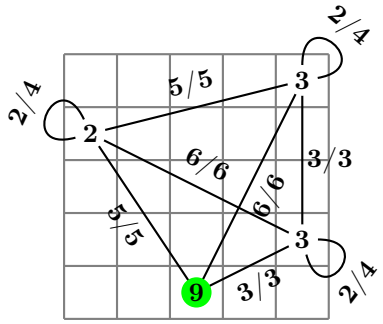


Abbildung 22: Abstraktion von `abbiegen0.txt`

3.2.2 Die Lösung finden

Mit dieser Abstraktion wird nun der Bruteforce vom Startknoten aus gestartet. Hier werden in der folgenden Abbildung 23 einige Iterationen gezeigt. Dort sieht man den Zustand des derzeitigen Pfades und die Prioritätswarteschlange abgebildet. Im Spielfeld sind die neuen Pfade zu den benachbarten Batterien abgebildet. Die Liste an Zahlen über den Kanten sind die Distanzen, die gegangen wurden. In der Prioritätswarteschlange ist zuerst die Punktzahl und dann der letzte Zug aller Pfade dargestellt. Bei unlösbaren Konfigurationen werden die Gruppen farblich gekennzeichnet. Nach jeder Iteration wird der nächste Pfad der oberste in der Warteschlange, so ist der Pfad {64: (5|4) 9} in der ersten Iteration der dargestellte Zustand in der zweiten Iteration.

In Abbildung 23a gehen vom Roboter neun neue Pfade aus, wie in der Warteschlange zu sehen ist. Dabei erkennt man gut, wie die einzelnen Pfade sortiert werden. Die Pfade, die eine Distanz von 9 Feldern gegangen sind, sind an erster Stelle in der Warteschlange. Dabei wird das Feld (1|2) immer schlechter eingestuft, als (5|4), da dieses weiter vom Hamiltonkreis abweicht. Der Zug zum Feld (5|1) mit einer Distanz von 8 ist nicht in der Warteschlange zu erkennen, da dieses viel zu weit vom Hamiltonkreis abweicht.

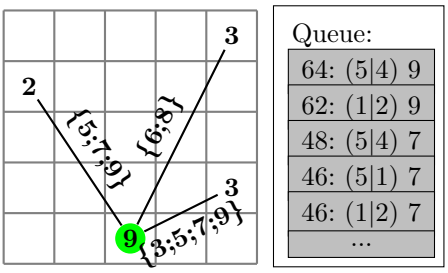
Hieran kann man auch gut erkennen, dass der Algorithmus auf weniger dichten Spielfeldern eher einem Greedy-Algorithmus ähnelt. Auf Spielfeldern, wie `stromralley1.txt` und `stromralley2.txt` hingegen fällt die Abweichung vom Hamiltonkreis viel stärker ins Gewicht, wodurch dort ein logischer Pfad durch alle Knoten bevorzugt wird und der Algorithmus auch sehr schnell zum Ziel kommt.

In den folgenden Abbildung 23b bis 23d werden nun all diese Pfade abgearbeitet, aber keiner wird fortgeführt, da die Gruppenunterteilung die Spielsituationen als unlösbar einstuft. Erst beim letzten Pfad in der Warteschlange wird die Spielsituation nicht unlösbar und zwei neue Pfade können generiert werden. (siehe Abb. 23e)

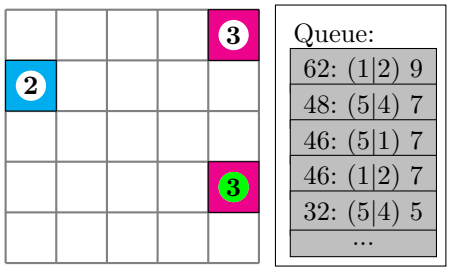
In den darauffolgenden Iterationen sind nurnoch wenige Möglichkeiten vorhanden und der Algorithmus entscheidet sich immer für den richtigen Zug. So wird nun am Ende in Abbildung 23h nurnoch die Raute um den Roboter herum auf ein freies Feld untersucht und die Lösung zurückgegeben.

3.2.3 Die Schrittfolge konstruieren

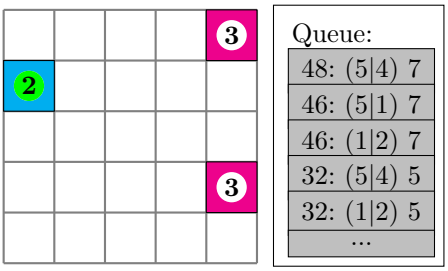
Aus der Zugfolge wird nun die Schrittfolge rekonstrutiert. Dazu werden alle Züge durchiteriert, bis auf den letzten Zug da dieser das freie Feld um den Roboter speichert und somit keinen Pfad in der Adjazenzliste hat. Für jeden Zug wird der zugehörige Pfad abgerufen und zur Schrittfolge hinzugefügt. (siehe Abb. 24)



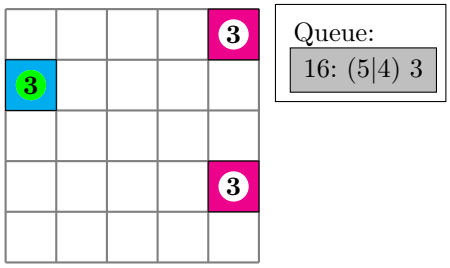
(a) 9 neue Pfade wurden generiert



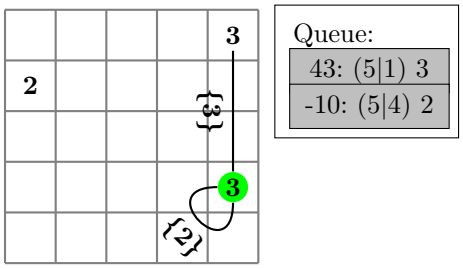
(b) Unlösbare Konfiguration wird verworfen



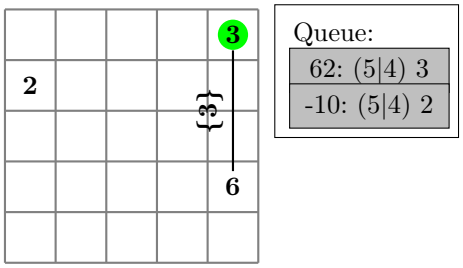
(c) Unlösbare Konfiguration wird verworfen



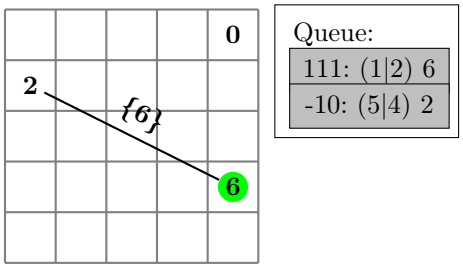
(d) Einige Iterationen später



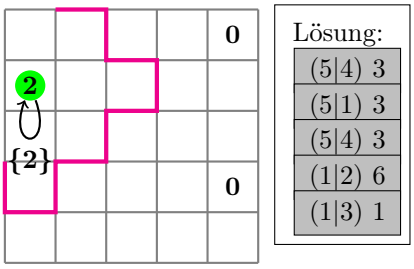
(e) 2 neue Pfade wurden generiert



(f) Ein neuer Pfad wurde generiert



(g) Ein neuer Pfad wurde generiert



(h) Raute wird überprüft und Ergebnis zurückgegeben

Abbildung 23: Iterationen des Bruteforce Algorithmus

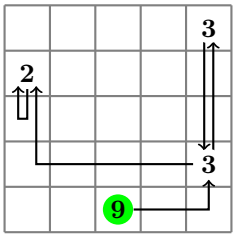


Abbildung 24: Lösungsweg für `stromralley0.txt`

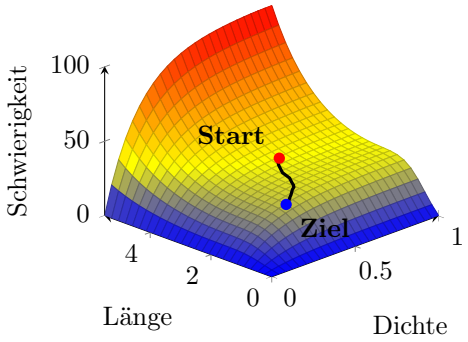


Abbildung 25: Gradient-Descent Beispiel

				0
	◯	**0**		
0	**0**			
	0			
	0	**0**	**0**	

Abbildung 26: Generiertes gelöstes Spielfeld

3.3 Beispiel der Erstellung von Spielsituationen

Hier wird nun näher erläutert, wie eine Spielsituation aus einem Schwierigkeitsgrad erstellt wird. Dazu wird eine Spielsituation mit 5x5 großen Spielfeld und den Standardwerten generiert.

3.3.1 Lösen der Bedingungen

Zuerst wird ein Zielschwierigkeitsgrad zufällig im Bereich von 0 bis 100 gewählt. Dieser ist in diesem Fall 30. Daraufhin wird vom Mittelpunkt der Funktion nun per Gradient-Descent zum Zielschwierigkeitsgrad gegangen. Am Ende bricht der Algorithmus mit einem Schwierigkeitsgrad von 29,96 ab, da dieser innerhalb der Toleranz von 0,1 liegt. Dieser Schwierigkeitsgrad entspricht hier einer Batteriedichte von 0,27 und einer Lösungslänge von 1,01. (siehe Abb. 25)

3.3.2 Konstruktion der Spielsituation

Nun wird mit diesen Spielfeldeigenschaften die Spielsituation konstruiert. Dazu wird zuerst die echte Batterieanzahl und echte Lösungslänge berechnet. Die Batterieanzahl ergibt sich aus der Multiplikation der Anzahl an Feldern und der Batteriedichte.

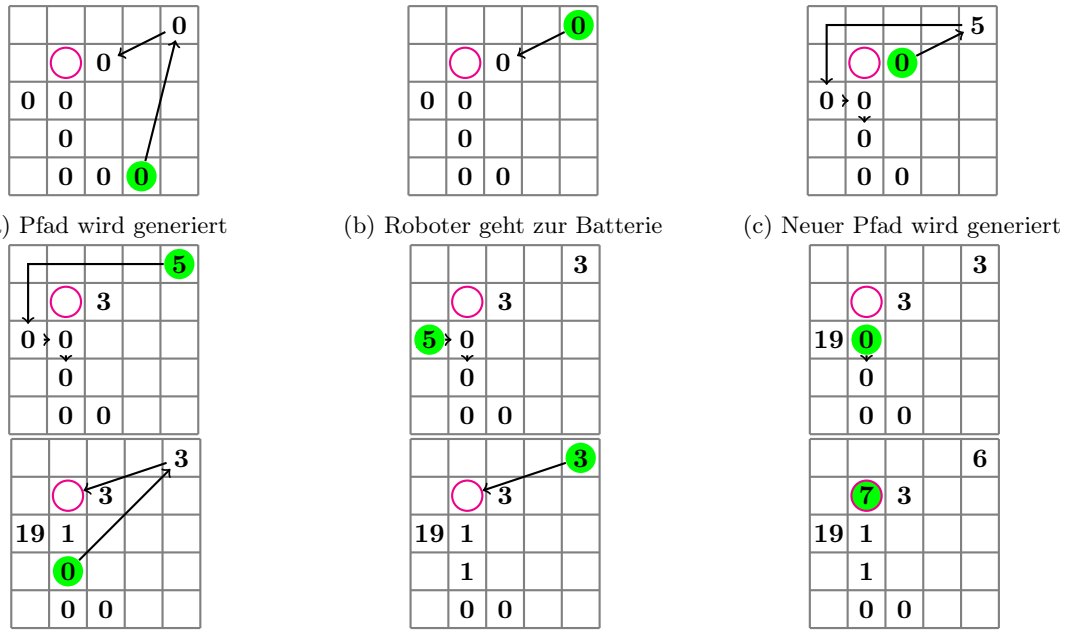
$$25 * 0,27 \approx 7$$

Multipliziert man die Batterieanzahl mit der Lösungslängenfaktor erhält man die Lösungslänge.

$$7 * 1,01 \approx 7$$

Diese Werte bedeuten, dass das Spielfeld 7 Batterien hat und man zu etwa 7 Batterien fahren muss bevor die Spielsituation gelöst ist. Die Start- und Endpositionen zusammen mit den Batterien werden nun zufällig auf dem Spielfeld verteilt. (siehe Abb. 26)

Nun wird der abstrahierte Graph dieses Spielfeldes generiert. Dieser Graph ist ein kompletter Graph, da jede Batterie von jeder anderen Batterie erreichbar ist. Deshalb ist der Graph hier nicht dargestellt. Auf dem Graphen wird nun die Batterie mit der niedrigsten Ladung ausgewählt. Da alle Batterien leer sind ist dies eine zufällige Batterie. Zu dieser Batterie wird ein Pfad gesucht und gegangen. (siehe Abb. 27)



(a) Pfad wird generiert (b) Roboter geht zur Batterie (c) Neuer Pfad wird generiert

Abbildung 27: Generieren einer Spielsituation

4 Quellcode

4.1 Struct Definitionen (global.hpp)

```
typedef struct path_t {
  bool available;                 // Pfad vorhanden?
  std::vector<point_t> shortest; // Kuerzester Pfad
  uint32_t length;                // Dessen Laenge

  bool extendable;                // Verlaengerbar?
  std::vector<point_t> extended; // Verlaengerbarer Pfad
  uint32_t extendedLength;        // Dessen Laenge
} path_t;

typedef struct robot_t {
  uint32_t position;
  uint32_t charge;
} robot_t;

typedef std::map<uint32_t, std::map<uint32_t, std::shared_ptr<path_t>>> adjacency_t;

typedef struct state_t {
  robot_t robot;
  std::map<uint32_t, uint32_t> batteries;
  size_t chargeSum;
} state_t;

typedef struct map_t {
  robot_t robot;
  std::map<uint32_t, uint32_t> batteries;
  size_t chargeSum;
  uint32_t size, area; // Groesse des Spielfelds
  adjacency_t adjacency; // Adjazenzliste
} map_t;
```

4.2 Spielfeldabstraktion (global.cpp)

```
uint32_t encode(point_t p, uint32_t size) {
  return (p.x - 1) + (p.y - 1) * size;
}

point_t decode(uint32_t n, uint32_t size) {
  return { (int)(n % size) + 1, (int)(n / size) + 1 };
```

```
}

// Abstrahiert das Spielfeld zu einem Graphen
void parseGraph(map_t &map) {
  map.adjacency = adjacency_t();
  std::map<uint32_t, uint32_t>::iterator u, v;

  point_t robot = decode(map.robot.position, map.size);

  // Fuer jede Batterie ...
  for (u = map.batteries.begin(); u != map.batteries.end(); ++u) {
    point_t pU = decode(u->first, map.size);
    map.adjacency[u->first] = std::map<uint32_t, std::shared_ptr<path_t>>();

    // ... zu jeder Batterie ... (auch sich selbst)
    for (v = map.batteries.begin(); v != map.batteries.end(); ++v) {
      point_t pV = decode(v->first, map.size);

      if (map.adjacency.find(v->first) == map.adjacency.end())
        map.adjacency[v->first] = std::map<uint32_t, std::shared_ptr<path_t>>();

      // ... einen Pfad generieren.
      auto path = findPath(pU, pV, map);
      map.adjacency[u->first][v->first] = path;
      map.adjacency[v->first][u->first] = path;
    }

    // Und auch zum Roboter
    auto path = findPath(pU, robot, map);
    map.adjacency[u->first][map.robot.position] = path;
    map.adjacency[map.robot.position][u->first] = path;
  }

  // Zur Vollstaendigkeit halber auch vom Roboter zum Roboter
  auto path = findPath(robot, robot, map);
  map.adjacency[map.robot.position][map.robot.position] = path;
}

// Findet den kuerzesten und verlaengerbaren Pfad zw. zwei Knoten
std::shared_ptr<path_t> findPath(point_t start, point_t end, map_t &map) {
  std::vector<point_t> shortest;
  // Schleifenpfad ist gefragt
  if (start == end) {
    // Freie Nachbarfelder suchen
    point_t dir = { 0, 1 };
    for (int i = 0; i < 4; i++, dir.rotate90()) {
      // Nachbarfeld
      point_t pos = start + dir;
      if (!pos.bounded(1, 1, map.size, map.size))
        continue;

      // Ist es frei?
      if (map.batteries.find(encode(start + dir, map.size)) == map.batteries.end()) {
        // Pfad generieren
        shortest = { start, start + dir, start };
        break;
      }
    }
  }
  else // Ansonsten BFS zum Pfad suchen benutzen
    shortest = BFS(start, end, map, false);

  std::shared_ptr<path_t> path = std::make_shared<path_t>();

  path->available = shortest.size() >= 2;
  path->shortest = shortest;
  path->length = shortest.size() - 1;

  // Pfad nicht verfuegbar?
  if (!path->available)
    return path;
  else if (path->length > 2) { // Schon verlaengerbar?
    path->extendable = true;
```

```
    path->extended = path->shortest;
    path->extendedLength = path->length;
    return path;
  }

  std::vector<point_t> extended;

  // Verlaengerbarer Schlaufenpfad gefragt
  // Also komplette Raute untersuchen
  if (start == end) {
    // Nachbarfelder untersuchen
    point_t dir = { 0, 1 };
    for (int i = 0; i < 4; i++, dir.rotate90()) {
      if (extended.size() > 0)
        break;

      // Nachbarfeld
      point_t middle = start + dir;
      if (!middle.bounded(1, 1, map.size, map.size))
        continue;

      if (map.batteries.find(encode(middle, map.size)) != map.batteries.end())
        continue;

      // Randfelder der Raute untersuchen
      for (int j = 0; j < 4; j++, dir.rotate90()) {
        // Randfeld der Raute
        point_t last = middle + dir;
        if (!last.bounded(1, 1, map.size, map.size))
          continue;

        // Ist es frei?
        if (map.batteries.find(encode(last, map.size)) == map.batteries.end()) {
          // Verlaengerbaren Pfad generieren
          extended = { start, middle, last, middle, start };
          break;
        }
      }
    }
  }
  else // Ansonsten wieder BFS benutzen
    extended = BFS(start, end, map, true);

  // Pfad nicht verfuegbar?
  if (extended.size() < 2)
    return path;

  path->extendable = true;
  path->extended = extended;
  path->extendedLength = extended.size() - 1;

  return path;
}

// Sucht einen Pfad zw. zwei Knoten
std::vector<point_t> BFS(point_t start, point_t goal, map_t &map, bool extendable) {
  std::queue<point_t> queue;
  queue.push(start);

  std::vector<bool> visited = std::vector<bool>(map.size * map.size);
  visited[encode(start, map.size)] = true;

  std::map<point_t, point_t> parent;

  // Hoechstens zwei zu ignorierende Kanten
  std::pair<point_t, point_t> skip1 = { };
  std::pair<point_t, point_t> skip2 = { };

  // Verlaengerbarer Pfad gefragt?
  // Zu blockierende Kanten suchen
  if (extendable) {
    // Raute ueberprufen
    point_t parallel = { 0, 1 }; // Parallele Felder
```

```
153     point_t diagonal = { 1, 1 }; // Diagonale Felder

155     for (int i = 0; i < 4; i++, parallel.rotate90(), diagonal.rotate90()) {
          // Zu blockierende Kanten ueberpruefen
157       if (start + parallel == goal) { // Direkt benachbart?
            // Eine Kante blockieren
159         skip1 = { start, goal };
            break;
161       }
          else if (start + parallel * 2 == goal) { // Zwei Felder entfernt?
163         // Eine Kante blockieren
            skip1 = { start + parallel, goal };
165         break;
          }
167       else if (start + diagonal == goal) { // Diagonal zur Batterie?
            // zwei Kanten blockieren
169         skip1 = { start + point_t{ diagonal.x, 0 }, goal };
            skip2 = { start + point_t{ 0, diagonal.y }, goal };
171         break;
          }
173     }
    }
175
    // Normalen BFS durchfuehren
177 while(!queue.empty()) {
      point_t u = queue.front();
179   queue.pop();
      if (u == goal)
181     break;

183   // Benachbarte Felder untersuchen
      point_t dir = { 1, 0 };
185   for (int i = 0; i < 4; i++, dir.rotate90()) {
        point_t v = u + dir;
187     uint32_t nodeV = encode(v, map.size);

189     // Ist die Kante blockiert worden?
        std::pair<point_t, point_t> edge = { u, v };
191     if (edge == skip1 || edge == skip2)
          continue;
193
        // Ist das Feld belegt oder schon besucht worden?
195     if (!v.bounded(1, 1, map.size, map.size)) continue;
        if (visited[nodeV]) continue;
197     if (v != goal && map.batteries.find(nodeV) != map.batteries.end()) continue;

199     // Feld als besucht markieren und Vorgaengerkarte erneuern
        visited[nodeV] = true;
201     parent[v] = u;
        queue.push(v);
203   }
    }
205
    std::vector<point_t> path;
207
    // Kein Pfad vorhanden?
209 if (parent.find(goal) == parent.end())
      return path;
211
    // Pfad aus Vorgaengerkarte generieren
213 point_t u = goal;
    while(u != start) {
215   path.push_back(u);
      u = parent[u];
217 }
    path.push_back(start);
219
    return path;
221 }
```

4.3 Struct Definitionen (solver.cpp)

```
1 // Repraesentiert einen
```

```
// Knoten im Hamiltonkreis
typedef struct node_t {
  node_t *prev;
  node_t *next;

  uint32_t point;

  ~node_t();

  int distance(uint32_t p);
  node_t* find(uint32_t p);
} node_t;

// Repraesentiert einen Zug
// im Pfad
typedef struct distance_pair_t {
  uint32_t position; // Neue Position
  uint32_t distance; // Gegangene Strecke
} distance_pair_t;

// Repraesentiert einen Pfad
typedef struct worker_t {
  std::vector<distance_pair_t> path;
  int score;
  int prevDirection = 0;

  void calculateScore(state_t &state);
  bool operator<(const worker_t &rhs) const;
} worker_t;
```

4.4 Generieren des Hamiltonkreises (solver.cpp)

```
// Bildet den Hamiltonkreis bei ungerader Groesse
static node_t* oddHamilton(node_t *start) {
  size_t count = 1; // Anzahl hinzugefügter Knoten

  std::vector<bool> visited = std::vector<bool>(map.area);
  visited[start->point] = true;

  point_t origin = decode(start->point, map.size);
  point_t pos = origin;
  node_t *current = start;

  // Ecke finden
  point_t corner;
  corner.x = pos.x <= map.size / 2 ? 1 : map.size;
  corner.y = pos.y <= map.size / 2 ? 1 : map.size;

  // Richtung zur Ecke bestimmen
  point_t dir = corner;
  dir.x -= map.size / 2;
  dir.y -= map.size / 2;
  dir.normalize();

  // Sicherstellen, dass dannach rotiert werden kann
  if (dir.x != dir.y) // Zuerst in y-Richtung gehen
    dir.x = 0;
  else                // Zuerst in x-Richtung gehen
    dir.y = 0;

  // Helfermethode fuegt einen Punkt zum Hamiltonkreis hinzu
  std::function<void(point_t)> addPoint = [&current, &visited, &count](point_t p) {
    current = (current->next = new node_t { current, NULL, encode(p, map.size) });
    visited[encode(p, map.size)] = true;
    count++;
  };

  // Zur Wand laufen (oder nicht im Edge-case)
  while ((dir.x && pos.x != corner.x && pos.y != 1 && pos.y != map.size)
    || (dir.y && pos.y != corner.y && pos.x != 1 && pos.x != map.size))
  {
    pos += dir;
    addPoint(pos);
```

```
  }

  // Zur Ecke laufen
  dir.rotate90();
  while((dir.x && pos.x != corner.x) || (dir.y && pos.y != corner.y)) {
    pos += dir;
    addPoint(pos);
  }

  dir.rotate90();
  pos += dir;

  point_t orth = dir; // Orthogonale bilden
  orth.rotate90();

  while(count < map.area) {
    addPoint(pos);

    // Ueber den Ursprung getreten?
    // --> Um 90 Grad drehen
    if (dir.x && (pos.x == origin.x + dir.x) && (pos.y == 1 || pos.y == map.size)) {
      dir.rotate90();
      orth.rotate90();
    }
    else if (dir.y && (pos.y == origin.y + dir.y) && (pos.x == 1 || pos.x == map.size)) {
      dir.rotate90();
      orth.rotate90();
    }

    // Entlang der Orthogonalen gehen
    point_t newPos = pos - orth;
    if (newPos.bounded(1, 1, map.size, map.size) && !visited[encode(newPos, map.size)]) {
      pos = newPos;
      continue;
    }

    newPos = pos + orth;
    if (newPos.bounded(1, 1, map.size, map.size) && !visited[encode(newPos, map.size)]) {
      pos = newPos;
      continue;
    }

    // Entlang der Blickrichtung gehen
    newPos = pos + dir;
    // Edge-case mit einen fehlenden Feld
    if (!newPos.bounded(1, 1, map.size, map.size)) {
      for (int missing = 0; missing < map.area; missing++) {
        if (!visited[missing]) {
          addPoint(decode(missing, map.size));
          break;
        }
      }
      break;
    }
    else if (!visited[encode(newPos, map.size)]) {
      pos = newPos;
      continue;
    }
  }

  return current;
}

// Bildet den Hamiltonkreis bei gerader Groesse
static node_t* evenHamilton(node_t *start) {
  point_t pos = decode(start->point, map.size);
  point_t origin = pos;

  do {
    // An der linken Wand entlang
    if(pos.x == 1 && pos.y < map.size) {
      pos.y += 1;
      start = (start->next = new node_t { start, NULL, encode(pos, map.size) });
```

```
115       continue;
      }
117
      // An der unteren Wand eintlang
119   if (pos.y == map.size) {
        if (pos.x < map.size)
121       pos.x += 1;
        else
123       pos.y -= 1;

125     start = (start->next = new node_t { start, NULL, encode(pos, map.size) });
        continue;
127   }

129   // Je nach Position nach oben
      // oder unten schlaengeln
131   if (pos.x % 2) {
        if (pos.y == map.size - 1)
133       pos.x -= 1;
        else
135       pos.y += 1;

137     start = (start->next = new node_t { start, NULL, encode(pos, map.size) });
      }
139   else {
        if (pos.y == 1)
141       pos.x -= 1;
        else
143       pos.y -= 1;

145     start = (start->next = new node_t { start, NULL, encode(pos, map.size) });
      }
147 } while (pos != origin);

149 // Pointer ist wieder am Anfang, so waere
    // der Anfang zweimal enthalten
151 start = start->prev;
    delete start->next;
153 return start;
}
155
// Bildet den Hamiltonkreis der Spielsituation
157 static void findHamilton() {
      hamiltonCycle = new node_t { NULL, NULL, map.robot.position };
159
    if (map.size % 2)
161   hamiltonCycle->prev = oddHamilton(hamiltonCycle);
    else
163   hamiltonCycle->prev = evenHamilton(hamiltonCycle);

165 hamiltonCycle->prev->next = hamiltonCycle;
}
```

4.5 Punktzahlberechnung (solver.cpp)

```
    // Berechnet die Punktzahl des Pfades
2   // Beachte, dass der State noch auf dem alten Stand ist
    void worker_t::calculateScore(state_t &state) {
4     // Entladene Ladung berechnen
      score = map.chargeSum - (state.chargeSum - this->path.back().distance);
6     score *= map.area / state.batteries.size(); // Ladungswert skalieren

8     // Hamiltonkreis zur alten Position rotieren
      hamiltonCycle = hamiltonCycle->find(state.robot.position);
10
      // Abweichung zur neuen Position berechnen
12    int signedDistance = hamiltonCycle->distance(this->path.back().position);
      int sign = sgn(signedDistance);
14
      // Fand ein Richtungswechsel statt?
16    if (sign != prevDirection) {
        // "Strafe" fuers umdrehen
18      signedDistance *= 2;
```

```
    prevDirection = sign;
  }

  // Abweichung von der Punktzahl abziehen
  score -= std::abs(signedDistance);
}
```

4.6 Gruppenüberprüfung (solver.cpp)

```
// Testet ob die Batterien in getrennten Gruppen gebuendelt sind.
// Dies macht naemlich den Zustand unloesbar.
static bool checkClique(state_t &state) {
  // Stellt eine Gruppe dar
  typedef std::pair<uint32_t, std::vector<uint32_t>> group_t;

  // Gruppenliste mit erster Gruppe initialisieren
  std::vector<group_t> groups = {{state.robot.charge, {state.robot.position}}};

  for (auto battery : state.batteries) {
    // Leere Batterien ignorieren
    if (battery.second == 0)
      continue;

    // Alle zugehoerigen Gruppen der Batterie
    std::vector<int> foundGroups;

    for (int i = 0; i < groups.size(); i++) {
      // Alle Batterien in der Gruppe durchgehen
      for (auto member : groups[i].second) {
        auto path = map.adjacency[battery.first][member];

        if (path->available) {
          // Pfadlaenge kleiner als Batterieladung
          // oder maximale Gruppenladung
          if (path->length <= battery.second || path->length <= groups[i].first) {
            foundGroups.push_back(i);
            break;
          }
        }
      }
    }

    // Gruppen zusammenfuehren
    if (foundGroups.size() > 1) {
      group_t newGroup = groups[foundGroups[0]];

      // Zusammenfuehren
      for (int i = 1; i < foundGroups.size(); i++) {
        group_t otherGroup = groups[foundGroups[i]];
        newGroup.second.insert(
          newGroup.second.end(),
          otherGroup.second.begin(),
          otherGroup.second.end()
        );

        if (newGroup.first < otherGroup.first)
          newGroup.first = otherGroup.first;
      }

      // Batterie hinzufuegen
      newGroup.second.push_back(battery.first);

      // Rekordladung erneuern
      if (newGroup.first < battery.second)
        newGroup.first = battery.second;

      // Alte Gruppe loeschen
      uint32_t deleted = 0;
      for (auto index : foundGroups)
        groups.erase(groups.begin() + index - (deleted++));

      // Neue Gruppe hinzufuegen
      groups.push_back(newGroup);
```

```
65      }
        // Zur Gruppe hinzufuegen
67      else if (foundGroups.size() == 1) {
          // Batterie hinzufuegen und Rekordladung erneuern
69        groups[foundGroups[0]].second.push_back(battery.first);
          if (groups[foundGroups[0]].first < battery.second)
71          groups[foundGroups[0]].first = battery.second;
        }
73      else // Neue Gruppe erstellen
          groups.push_back({ battery.second, { battery.first }});
75    }

77    // Eine Grosse Gruppe vorhanden?
      return groups.size() == 1;
79  }
```

4.7 Bruteforce Implementation (solver.cpp)

```
    // Rekonstruiert das Spielfeld aus der Zugfolge
2   static state_t reconstructState(worker_t &path) {
      // Ausganssituation kopieren
4     state_t state = {map.robot, map.batteries, map.chargeSum};

6     for (auto move : path.path) {
        // Zug durchfuehren
8       uint32_t oldCharge = state.robot.charge - move.distance;
        state.robot.charge = state.batteries[move.position];
10      state.batteries[move.position] = oldCharge;

12      state.chargeSum -= move.distance;
      }
14
      // Position des Roboters setzten
16    if (path.path.size() > 0)
        state.robot.position = path.path.back().position;
18
      return state;
20  }

22  // Implementation des Bruteforce-Heuristik Algorithmus
    static std::pair<worker_t, state_t> solveConfig(bool debug)
24  {
      // Prioritaetswarteschlange an Pfaden
26    std::priority_queue<worker_t> queue;
      queue.push(worker_t());
28
      if (debug)
30      std::cout << "Iterationen:" << std::endl;

32    while(!queue.empty()) {
        worker_t worker = queue.top();
34      queue.pop();

36      // Spielfeld rekonstruieren
        state_t state = reconstructState(worker);
38
        // Loesbarkeit ueberprufen
40      if (!checkClique(state))
          continue;
42
        if (debug)
44        std::printf("Score:␣%-5i␣rem.␣charge:␣%-5lu\n", worker.score, state.chargeSum);

46      // Alle Batterien leer?
        if (state.chargeSum - state.robot.charge == 0) {
48        if (state.robot.charge == 0)
            return { worker, state }; // Schon fertig
50
          // Pfad generieren
52        point_t robotPos = decode(state.robot.position, map.size);
          auto path = findPath(robotPos, robotPos, map);
54
          if (state.robot.charge == 2) {
```

```
56          if (path->available) {
              worker.path.push_back({ encode(path->shortest[1], map.size), 1});
58            return {worker, state};
            }
60        }
          else if (state.robot.charge > 2) {
62          if (path->extendable) {
              worker.path.push_back({ encode(path->shortest[1], map.size), 1});
64            worker.path.push_back({ encode(path->shortest[2], map.size), 1});
              return {worker, state};
66          }
          }
68        else {
            // Umliegendes Feld suchen
70          point_t dir = { 0, 1 };
            for (int i = 0; i < 4; i++, dir.rotate90()) {
72            point_t middle = robotPos + dir;
              if (!middle.bounded(1, 1, map.size, map.size))
74              continue;

76            worker.path.push_back({ encode(middle, map.size), 1});
              return {worker, state};
78          }
          }
80
          continue; // Doch keine Loesung
82      }

84      // Benachbarte Batterien durchlaufen
        for (auto neighbour : map.adjacency[state.robot.position]) {
86        if (!neighbour.second->available) //Pfad nicht verfuegbar?
            continue;
88        if (neighbour.first == map.robot.position) // Keine Batterie?
            continue;
90        if (state.batteries[neighbour.first] == 0) // Batterie ist leer?
            continue;
92
          // Laenge des kuerzesten Pfades
94        uint32_t min = neighbour.second->length;
          if (state.robot.charge < min) continue;
96
          // Kuerzesten Pfad gehen
98        worker_t newWorker = worker;
          newWorker.path.push_back({neighbour.first, min});
100       newWorker.calculateScore(state);
          queue.push(newWorker);
102
          // Eventuell den verlaengerbaren Pfad gehen
104       if (neighbour.second->extendable) {
            min = neighbour.second->extendedLength;
106         uint32_t max = state.robot.charge;

108         // Denselben Pfad nicht zweimal gehen
            if (min == neighbour.second->length)
110           min += 2;

112         // Pfad kann immer um 2 verlaengert werden
            // Entspricht dann einmal hin und her fahren
114         for (uint32_t distance = min; distance <= max; distance += 2) {
              // Verlaengerten Pfad gehen
116           worker_t newWorker = worker;
              newWorker.path.push_back({ neighbour.first, distance });
118           newWorker.calculateScore(state);
              queue.push(newWorker);
120         }
          }
122     }
    }
124
    if (debug)
126   std::cout << '\n' << std::endl;

128 return {};
```

```
}
```

4.8 Rekonstruktion der Schrittfolge (solver.cpp)

```
// Rekonstruiert die Schrittfolge aus der Zugfolge
static std::vector<point_t> constructPath(worker_t &solution, state_t &state) {
  // Keine Loesung
  if (solution.path.size() == 0)
    return {};

  std::vector<point_t> path;

  for(int i = 0; i < solution.path.size(); i++) {
    // Letzte 2 Punkte im Pfad geben die freien Felder um den Roboter an
    if (state.robot.charge > 2 && i >= solution.path.size() - 2)
      break;

    // Letzer Punkt im Pfad gibt das freie Feld beim Roboter an
    if (state.robot.charge > 0 && i >= solution.path.size() - 1)
      break;

    // Start- und Endknoten des Zuges abrufen
    uint32_t nodeA;
    if (i == 0) // Urspruengliche Position des Roboters nehmen
      nodeA = map.robot.position;
    else
      nodeA = solution.path[i - 1].position;

    uint32_t nodeB = solution.path[i].position;

    point_t pA = decode(nodeA, map.size);
    point_t pB = decode(nodeB, map.size);

    // Pfad zwischen beiden Knoten abrufen
    auto edge = map.adjacency[nodeA][nodeB];
    uint32_t distance = solution.path[i].distance;

    // Pfad zur Schrittfolge hinzufuegen.
    // Dabei immer den ersten Punkt ueberspringen,
    // da der Roboter dort schon ist.

    if (distance == edge->length) {
      // Kuerzesten Pfad kopieren
      // Muss der Pfad umgedreht werden?
      if (edge->shortest[0] == pA)
        path.insert(path.end(), edge->shortest.begin() + 1, edge->shortest.end());
      else
        // Pfad rueckwaerts einfuegen
        path.insert(path.end(), edge->shortest.rbegin() + 1, edge->shortest.rend());
    }
    else {
      // Verlaengerbaren Pfad entsprechend
      // verlaengern und einfuegen
      std::vector<point_t> extended;

      // Noetige Verlaengerung berechnen
      uint32_t extension = distance - edge->extendedLength;
      size_t end = edge->extended.size() - 1;

      // Verlaengerung zum Pfad hinzufuegen
      uint32_t current = 0;
      while(extension > 0) {
        if (edge->extended[0] == pA)
          extended.push_back(edge->extended[1 + current]);
        else
          extended.push_back(edge->extended[end - 1 - current]);

        current = (current + 1) % 2; // Immer hin und her
        extension--;
      }
      path.insert(path.end(), extended.begin(), extended.end());

      // Restlichen Pfad hinzufuegen
```

```
      if (edge->extended[0] == pA)
        path.insert(path.end(), edge->extended.begin() + 1, edge->extended.end());
      else
        path.insert(path.end(), edge->extended.rbegin() + 1, edge->extended.rend());
    }
  }

  if (state.robot.charge > 2) {
    // Letzte beiden Felder so oft wie noetig wiederholen
    size_t size = solution.path.size();
    int current = 1;
    while(state.robot.charge > 0) {
      path.push_back(decode(solution.path[size - 1 - current].position, map.size));
      current = (current + 1) % 2; // Immer hin und her gehen
      state.robot.charge--;
    }
  }
  else if (state.robot.charge > 0) {
    // Das freie Feld zum Pfad hinzufuegen
    path.push_back(decode(solution.path.back().position, map.size));

    // Eventuell wieder auf das vorherige Feld gehen
    if (state.robot.charge == 2)
      path.push_back(decode(state.robot.position, map.size));
  }

  return path;
}
```

4.9 Struct Definitionen (generator.hpp)

```
// Stellt die Begrenzungen für den Gradient-Descent dar
typedef struct constraint_t {
  uint32_t size = 10;
  double minDI = 0, maxDI = 100;
  double minDensity = 0, maxDensity = 1;
  double minLength = 0, maxLength = 5;
} constraint_t;

// Stellt die Spielfeldeigenschaften dar
typedef struct difficulty_t {
  uint32_t size;
  double density;
  double length;

  double di;
  double deltaDensity;
  double deltaLength;

  void calculate();
} difficulty_t;
```

4.10 Gradient-Descent Implementation (generator.cpp)

```
// Berechnet die Spielfeldeigenschaften aus den Beschraenkungen
difficulty_t solveConstraints(constraint_t constraints, bool debug) {
  std::uniform_real_distribution<double> distrib(constraints.minDI, constraints.maxDI);
  std::default_random_engine random;
  random.seed(std::rand());

  // Zielschwierigkeit waehlen
  double targetDI = distrib(random);

  // Mittelpunkt im erlaubten Bereich setzen
  difficulty_t diff;
  diff.size = constraints.size;
  diff.density = (constraints.minDensity + constraints.maxDensity) / 2.0;
  diff.length = (constraints.minLength + constraints.maxLength) / 2.0;
  diff.calculate();

  // Bis der Abstand zur Zielschwierigkeit klein genug ist
  size_t iterations = 0;
  while(iterations++ < MAX_ITERATIONS && std::abs(targetDI - diff.di) > MAX_DELTA) {
```

```
20      // Gradient-Descent anwenden
        diff.density -= diff.deltaDensity * D_DELTA * sgn(diff.di - targetDI);
22      diff.length -= diff.deltaLength * L_DELTA * sgn(diff.di - targetDI);

24      // Beim Austreten aus den Grenzen zurueckgehen
        if (diff.density < constraints.minDensity || diff.density > constraints.maxDensity)
26        diff.density += diff.deltaDensity * D_DELTA * sgn(diff.di - targetDI);

28      if (diff.length < constraints.minLength || diff.length > constraints.maxLength)
          diff.length += diff.deltaLength * L_DELTA * sgn(diff.di - targetDI);
30
        // Neue Funktionswerte berechnen
32      diff.calculate();

34      if (debug)
          std::printf("Iter. %-6lu  Target: %-3.2f  DI: %-3.2f  D: %-3.2f  L: %-3.2f \n",
36          iterations,
            targetDI,
38          diff.di,
            diff.density,
40          diff.length
          );
42    }

44    // Spielfeldeigenschaften zurueckgeben
      return diff;
46  }
```

4.11 Generieren der Spielsituation (generator.cpp)

```
    // Sucht einen Pfad durch die Spielfeldabstraktion von einer Batterie zu einer anderen
2   static std::vector<uint32_t> batteryBFS(uint32_t start, uint32_t end, map_t &map) {
      // Eigenpfad gefragt und verfuegbar?
4     if (start == end && map.adjacency[start][end]->available)
        return { end };
6
      std::queue<uint32_t> queue;
8     std::vector<bool> visited = std::vector<bool>(map.area, false);
      std::vector<uint32_t> parent = std::vector<uint32_t>(map.area, map.area);
10
      queue.push(start);
12
      while(!queue.empty()) {
14      uint32_t u = queue.front();
        queue.pop();
16
        if (u == end)
18        break;

20      if (visited[u])
          continue;
22
        visited[u] = true;
24
        // Nachbarn durchlaufen
26      for (auto v : map.adjacency[u]) {
          if (!v.second->available)
28          continue;

30        if (visited[v.first])
            continue;
32
          // Pfad soll nur ueber Batterien gehen abgesehen vom Start und Ziel
34        // Die Adjazenzliste beinhaltet naemlich auch Pfade zum Roboter
          if (v.first != start && v.first != end
36          && map.batteries.find(v.first) == map.batteries.end())
            continue;
38
          visited[v.first] = true;
40        parent[v.first] = u;
          queue.push(v.first);
42      }
    }
```

```

  // Batteriepfad konstruieren
  std::vector<uint32_t> path;
  uint32_t current = end;
  while(current != start)
  {
    if (current == map.area) { // Kein Pfad verfuegbar
      return {};
    }

    path.push_back(current);
    current = parent[current];
  }
  std::reverse(path.begin(), path.end());

  return path;
}

// Generiert eine Spielsituation aus den Spielfeldeigenschaften
map_t generateConfig(difficulty_t difficulty) {
  map_t map;
  map.size = difficulty.size;
  map.area = map.size * map.size;

  // Richtige Batterieanzahl und Loesungslaenge berechnen
  size_t batteryCount = (size_t)std::round((double)map.area * difficulty.density);
  size_t pathLength = (size_t)std::round((double)batteryCount * difficulty.length);

  // Batterieanzahl auf maximale Anzahl beschraenken
  if (batteryCount > map.area - 1)
    batteryCount = map.area - 1;

  std::vector<bool> free = std::vector<bool>(map.area, true);

  // Generiert ein zufaelliges Feld aus den noch freien Feldern
  std::function<uint32_t()> getRandom = [&free, &map](){
    // Zufaellig waehlen solange
    // gewaehltes belegt ist
    uint32_t choice;
    do {
      choice = std::rand() % map.area;
    } while(!free[choice]);
    free[choice] = false;

    return choice;
  };

  // Zaehlt wie oft die jeweiligen Batterien besucht wurden
  // std::unordered_map behaelt die zufaellige Verteilung
  std::unordered_map<uint32_t, uint32_t> batteryCounter;

  // Zielfeld aussuchen (Kann auf einer Batterie sein)
  uint32_t goalNode = std::rand() % map.area;

  // Startfeld aussuchen (Muss frei sein)
  uint32_t startNode = getRandom();

  // Batterien verteilen
  while(batteryCount--) {
    uint32_t battery = getRandom();
    map.batteries[battery] = 0;
    batteryCounter[battery] = 0;
  }

  // Adjazenzliste generieren
  map.robot.position = startNode;
  parseGraph(map);

  // Falls Zielknoten nicht auf einer Batterie liegt
  // muessen die Pfade zw. dieser und den Batterien generiert werden
  if (map.batteries.find(goalNode) == map.batteries.end()) {
    point_t goalPoint = decode(goalNode, map.size);
    for (auto battery : map.batteries) {
      auto path = findPath(decode(battery.first, map.size), goalPoint, map);
```

```
      map.adjacency[battery.first][goalNode] = path;
58    map.adjacency[goalNode][battery.first] = path;
    }
60
    auto path = findPath(decode(startNode, map.size), goalPoint, map);
62  map.adjacency[startNode][goalNode] = path;
    map.adjacency[goalNode][startNode] = path;
64  }

66  // Gibt die am wenigsten besuchte Batterie zurueck
    std::function<uint32_t()> getBattery = [&batteryCounter](){
68    std::pair<uint32_t, uint32_t> min = *batteryCounter.begin();

70    for (auto battery : batteryCounter)
        if (battery.second < min.second)
72        min = battery;

74    return min.first;
    };
76
    // Roboter auf das Zielfeld setzten
78  map.robot.position = goalNode;
    map.robot.charge = 0;
80
    // Solange der Roboter nicht am Startfeld ist
82  // (Ladung ueberprufen, falls das Startfeld auch
    // das Zielfeld ist)
84  while(map.robot.position != startNode || map.robot.charge == 0) {
      auto startPath = batteryBFS(map.robot.position, startNode, map);
86    std::vector<uint32_t> batteryPath;

88    // Pfadlaenge reicht nicht mehr, also zum Start laufen
      bool walkToStart = startPath.size() >= pathLength;
90
      if (walkToStart)
92      batteryPath = startPath;
      else // Ansonsten zu einer zufaelligen Batterie gehen
94      batteryPath = batteryBFS(map.robot.position, getBattery(), map);

96    while (batteryPath.size() == 0) { // Wenn kein Pfad gefunden wurde
        auto iter = map.batteries.begin();
98      std::advance(iter, rand() % map.batteries.size());
        batteryPath = batteryBFS(map.robot.position, (*iter).first, map);
100   }

102   // Fuer jede Batterie im Pfad
      for (auto battery : batteryPath) {
104     auto path = map.adjacency[map.robot.position][battery];
        if (!path->available)
106       break;

108     // Zufaellig kuerzesten oder laengeren Pfad
        // zwischen den Punkten waehlen
110     uint32_t distance = path->length;
        if (path->extendable && std::rand() % 10 > 7)
112       // Zufaellige Verlaengerung zw. 0 und 10 waehlen
          distance = path->extendedLength + (std::rand() % 6) * 2;
114
        // Pfad entlanglaufen (Distanz wird zur Ladung addiert)
116     if (battery == startNode)
          map.robot.charge += distance;
118     else {
          // Batterietausch
120       uint32_t oldRobotCharge = map.robot.charge;
          map.robot.charge = map.batteries[battery];
122       map.batteries[battery] = oldRobotCharge + distance;
        }
124
        // Roboter auf die Batterie setzen
126     map.robot.position = battery;

128     // Loesungslange verringern
        if (pathLength > 0)
```

```
130           pathLength--;

132         // Batteriezaehler erhoerhen
            batteryCounter[battery]++;
134       }
      }
136
      // Spielsituation ist fertig generiert
138   return map;
  }
```