Aufgabe 1: Stromralley

Teilnahme-Id: 9693

Bearbeiter/-in dieser Aufgabe:
Nick Djerfi

18. 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 rückwarts 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 bearbetet 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 von der vorherigen Position aus geprüft, wann die neue Position im logischen Pfad auftritt. 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. (siehe Abb. 1)

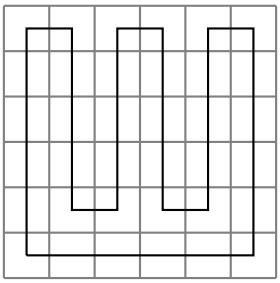
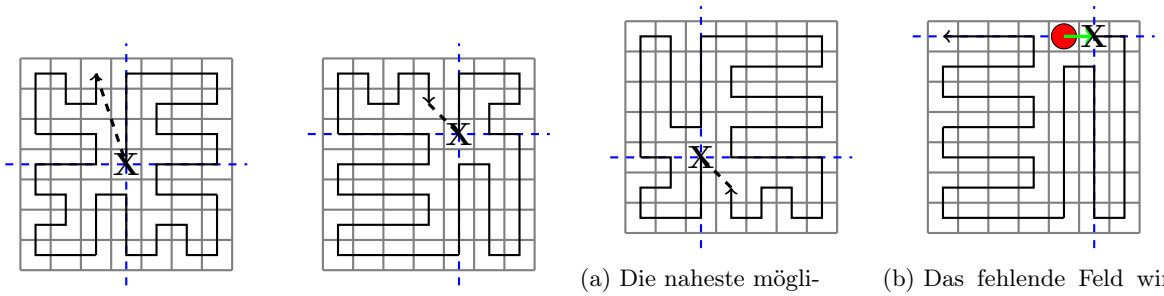


Abbildung 1: Hamiltonkreis bei geraden Spielfeldgrößen

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. 2)

In der folgenden Abbildung 3 sind einige Beispiele für die Abweichung vom Hamiltonpfad dargestellt.



(a) Die naheste mögliche Ecke wird gewählt

(b) Das fehlende Feld wird am Anfang angefügt

Abbildung 2: Hamiltonpfade bei ungeraden Seitenlängen

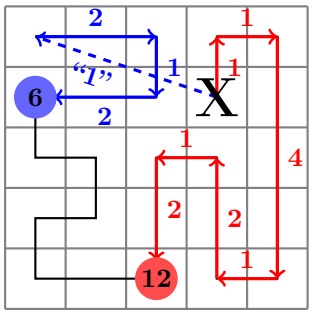


Abbildung 3: Abweichung vom Hamiltonpfad

1.3 Bound-Bedingung

Um unmögliche 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 Battere 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 gibt zur anderen Gruppe zu gehen. Natürlich werden hier leere Batterien ignoriert, bzw. als "Wände" gesehen.

(a) Eine große Gruppe

(b) Zwei kleine Gruppen

Abbildung 4: 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 5 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.

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

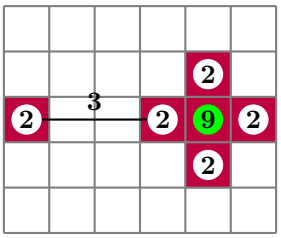


Abbildung 5: Edge-case

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, das 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. 6)

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

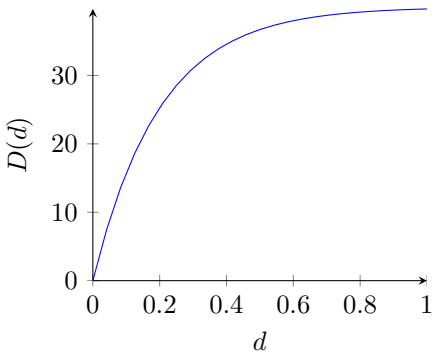


Abbildung 6: Schwierigkeitsgrad in Relation zur Batteriedichte

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. 7)

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

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

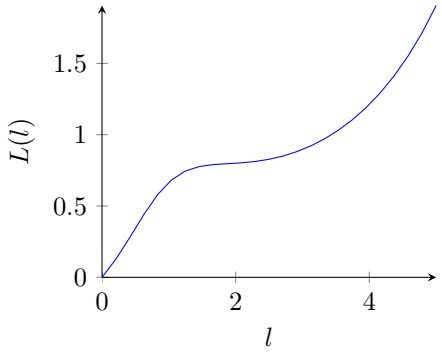


Abbildung 7: Schwierigkeitsgrad in Relation zur Lösungslänge

Einen Schwierigkeitsgrad erhält man aus der Kombination beider Funktionen. Dazu werden beide Funktionen miteinander multipliziert. (siehe Abb. 8)

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

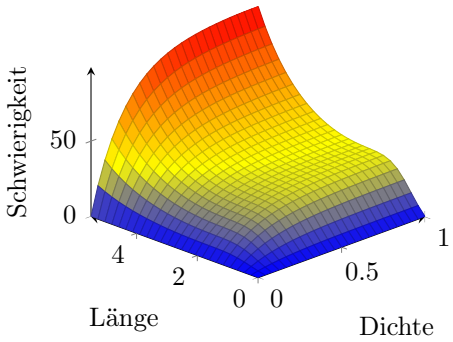


Abbildung 8: Schwierigkeitsgrad

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. 9)

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

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

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. 10)

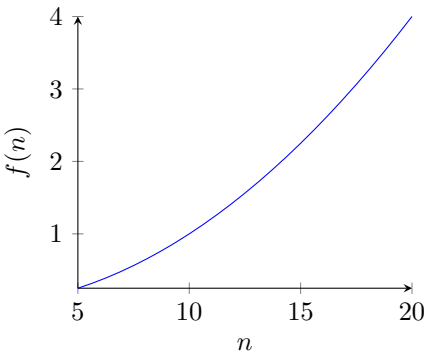


Abbildung 9: Schwierigkeitsgrad in Relation zur Spielfeldgröße

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

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

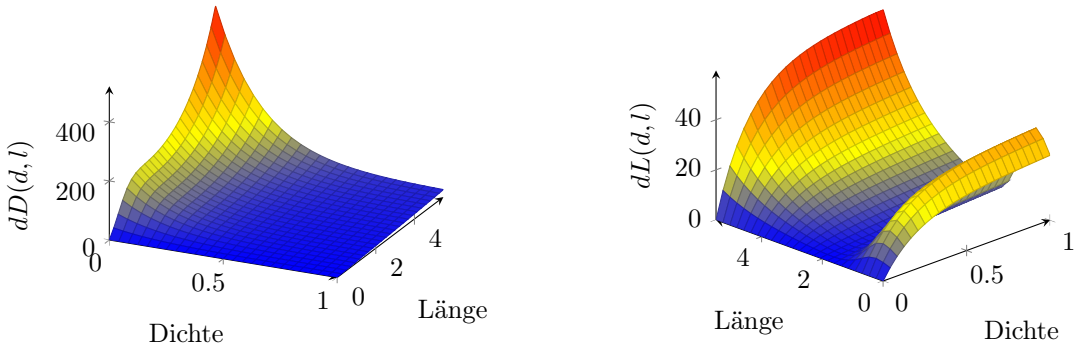


Abbildung 10: Ableitungsfunktionen

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. 11)

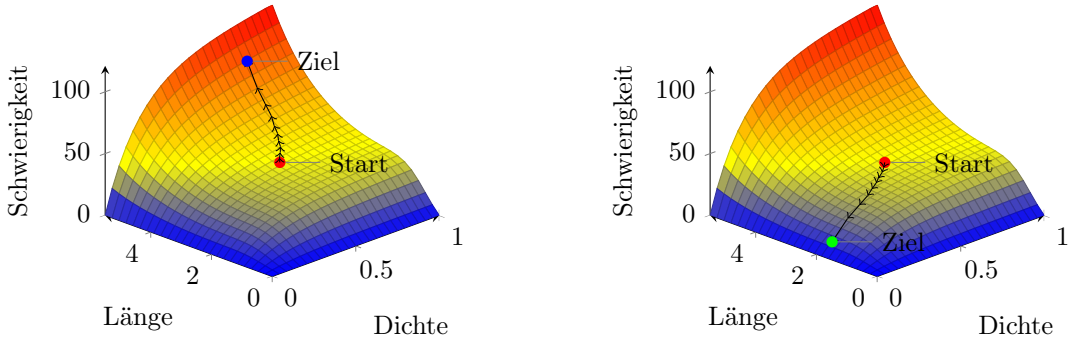


Abbildung 11: Gradient-Descent

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 Weg zur Startposition. Wenn der Roboter die Startposition erreicht hat, haben alle Batterien die richtige Ladung und der Roboter die richtige Position und Ladung. Zuletzt wird eine Datei für die Spielsituation erstellt. (siehe Abb. 12)

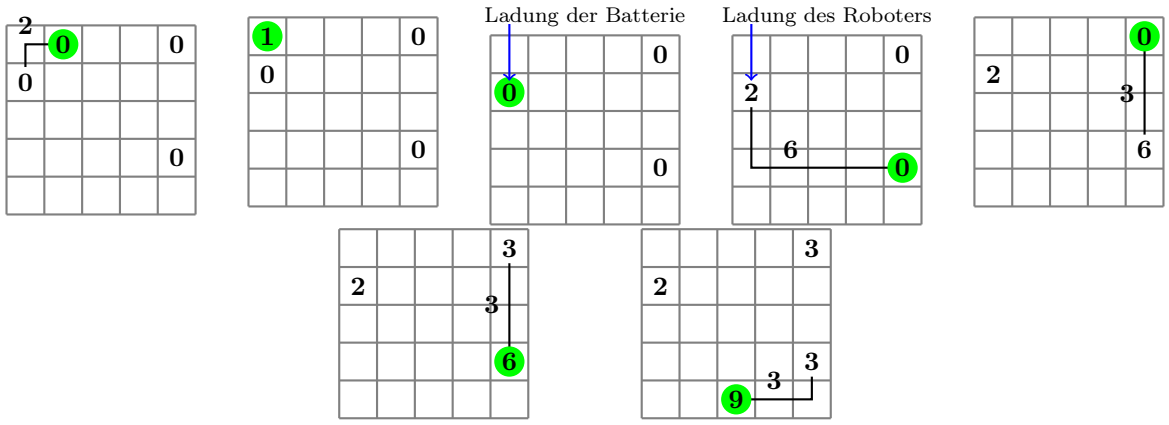


Abbildung 12: Generieren einer Spielsituation

2 Umsetzung

2.1 Übersetzten von Punkten zu Indizes

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. 13)

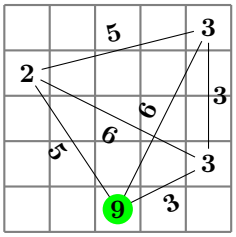


Abbildung 13: Abstraktion von `abbiegen0.txt`

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 14 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.

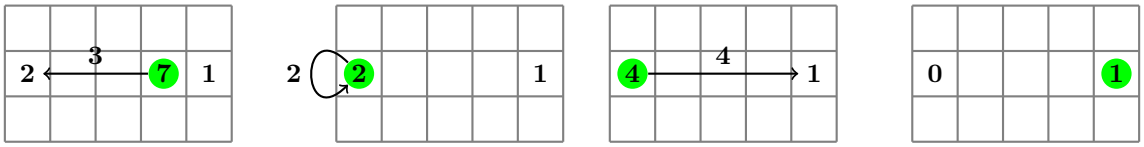
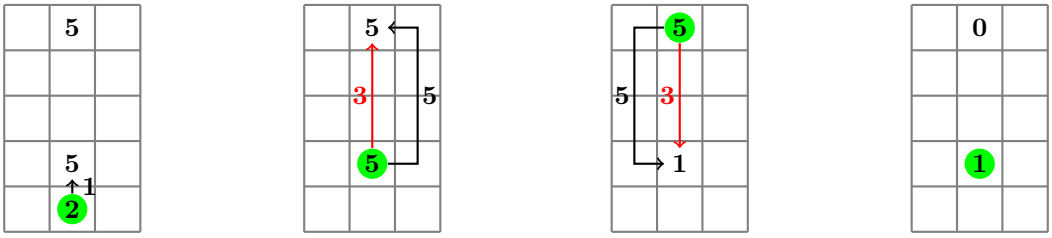


Abbildung 14: Beispiel von Schleifen

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 freien Feldern, auf denen der Roboter hin und her fahren kann, gespeichert werden.

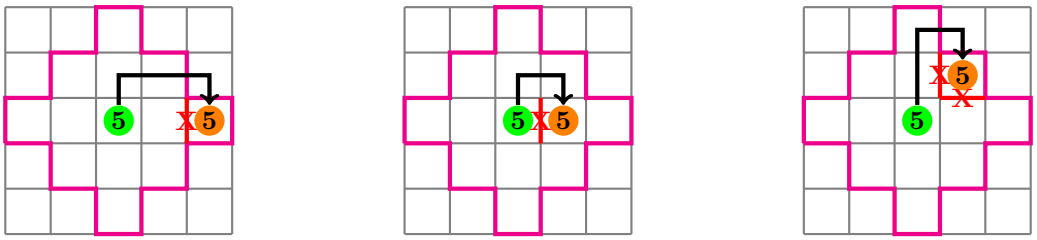


(a) Der Roboter hinterlässt eine Ladung von 1 (b) Der rote kürzeste Pfad macht es unlösbar (c) Der rote kürzeste Pfad macht es wieder unlösbar (d) Alle Batterien sind geleert

Abbildung 15: 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 Algorithmus diese nicht begeht. (siehe Abb. 16)



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

Abbildung 16: Rautenkonfigurationen

In einem Pfad Struct `path_t` 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.

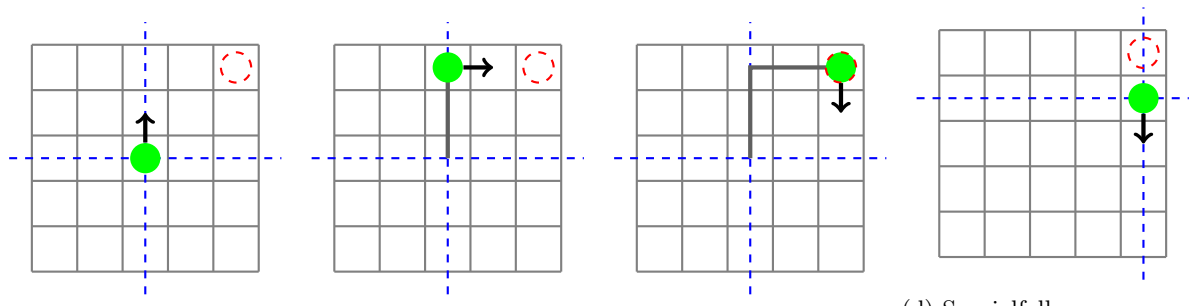
```
typedef struct path_t
{
  bool available;                 // Pfad vorhanden?
  std::vector<point_t> shortest; // Kuerzester Pfad
  uint32_t length;                // Dessen Laenge

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

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 bzw. 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.

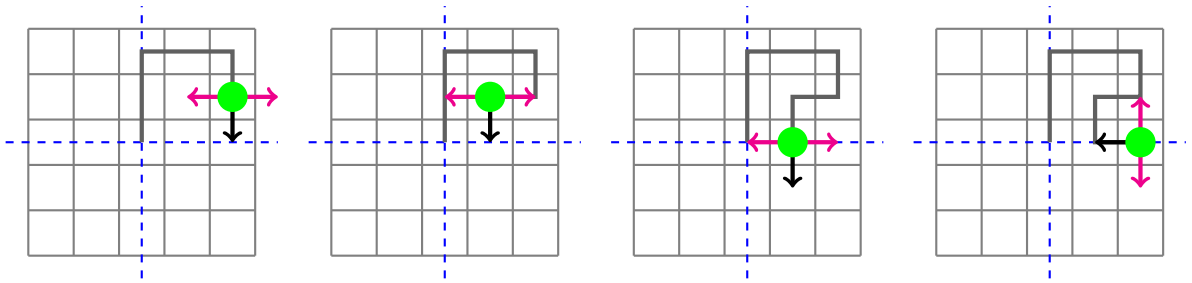


(a) Geht zur oberen Wand (b) Um 90° drehen (c) Geht zur Ecke (d) Spezialfall: Ecke unerreichbar

Abbildung 17: 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.

Dies wird solange wiederholt, bis alle Felder durchloffen wurden, oder der Algorithmus aus dem Spielfeld austritt. Der Algorithmus geht aus dem Spielfeld raus, wenn noch Felder fehlen. Es kann nämlich



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

Abbildung 18: Hin und her schlängeln

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 den Hamiltonkreis repräsentiert. 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 Gruppe zusammengefasst. Zuletzt wird die Batterie zu der Gruppe hinzugefügt. 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.

2.4.2 Bruteforce Algorithmus

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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.

```
score = (totalCharge - this->chargeSum) * (map.area / batteries.size());
```

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

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

// Abweichung zur letzten 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

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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 in Bezug auf die Variablen berechnet. Je nach dem, ob sich die Zielschwierigkeit über oder unter der 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)

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.

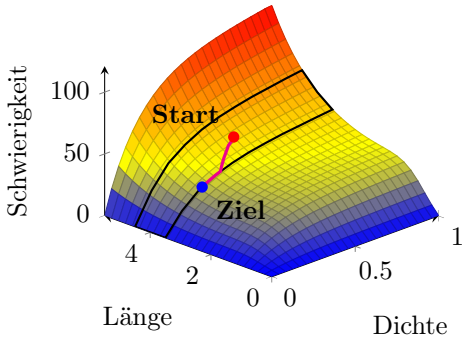
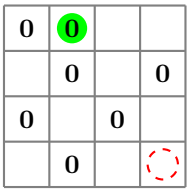
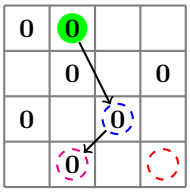


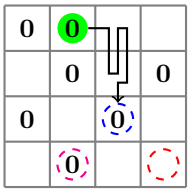
Abbildung 19: Gradient-Descent mit Grenzen



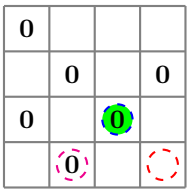
(a) Ausgangssituation



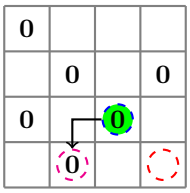
(b) Batteriepfad gewählt



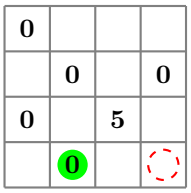
(c) Verlängerter Pfad gewählt



(d) Pfad gegangen



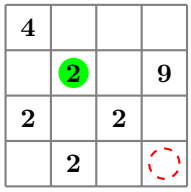
(e) Kürzesten Pfad gewählt



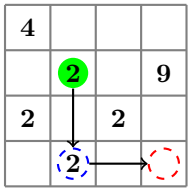
(f) Pfad gegangen

Abbildung 20: Rückwärts lösen

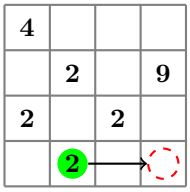
Ist die Lösungslänge kleiner als der Weg zur Startposition geht der Roboter sofort diesen Weg zur Startposition. Damit die Batterien gleichmäßig geladen werden, wird für jede Batterie gezählt, wie oft diese besucht wurde. Bei der Wahl der nächsten Batterie wird dann die am wenigsten besuchte gewählt.



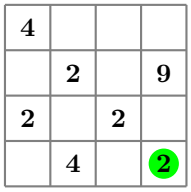
(a) Ausgangssituation



(b) Pfad zur Startposition



(c) Gehe zur Batterie



(d) Gehe zum Start

Abbildung 21: Zur Startposition gehen

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 Verzwigungsfaktor ist und d die Länge des Lösungsweges.

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 auch nahezu unmöglich werden.

TODO: Tabelle für Lösungsalgorithmus

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 1: 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

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.1 Ergebnisse

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2 Beispiel an `abbiegen0.txt`

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2.1 Spielfeldabstraktion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2.2 Die Lösung finden

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.3 Beispiel der Erstellung von Spielsituationen

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.3.1 Lösen der Bedingungen

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.3.2 Konstruktion der Spielsituation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

4 Quellcode