38. Bundeswettbewerb Informatik 2019/2020, 2. Runde

Lösungshinweise und Bewertungskriterien

Bundeswettbewerb Informatik

Allgemeines

Grundsätzliches

Es ist immer wieder bewundernswert, wie viele Ideen, wie viel Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur 2. Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Deswegen sind Punktabzüge die Regel und Bewertungen mit Pluspunkten über die Erwartungen hinaus die Ausnahme.

Spannende bzw. schwierige Erweiterungen der Aufgabenstellung sind immer einige Extrapunkte wert, wenn sie auch praktisch realisiert wurden. Intensive theoretische Überlegungen wie z. B. ein korrekter Beweis zur Komplexität des Problems werden ebenfalls mit zusätzlichen Punkten belohnt. Weitere Ideen ohne Implementierung und geringe Verbesserungen der bereits implementierten Lösung einer Aufgabe gelten allerdings nicht als geeignete Erweiterungen.

Wie auch immer Ihre Einsendung bewertet wurde, lassen Sie sich nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollten Sie nicht unterschätzen. Selbst wenn Sie zum Beispiel aus Zeitmangel nur die Lösung zu einer Aufgabe einreichten, so erhielten Sie eine Bewertung Ihrer Einsendung, die Ihnen bei der Anfertigung künftiger Lösungen hilfreich sein kann.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie bitte kurz die folgenden Anmerkungen zu den Einsendungen und beiliegenden Unterlagen durch.

Bewertungsbogen

Aus der ersten Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Auch in dieser Runde können Sie den Bewertungsbogen im Anmeldesystem PMS einsehen. In der ersten Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln dann abgezogen werden konnte. In der 2. Runde geht die Bewertung von zwanzig Punkten aus (es gibt nur ganze Punkte, keine halben). Im Vergleich zur 1. Runde gibt es deutlich mehr Bewertungskriterien, bei denen Punkte abgezogen oder auch hinzuaddiert werden konnten.

Terminlage

Für Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und zweiten Runde sicher nicht ideal. Doch leider bleibt dem Bundeswettbewerb Informatik nur die erste Jahreshälfte für die zweite Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des Mathematikwettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die zweite Runde beträgt etwa vier Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden.

Einige Aufgaben in der zweiten Runde sind oft deutlich schwerer zu lösen, als sie auf den ersten Blick erscheinen mögen. Erst in der konkreten Umsetzung einer Lösungsidee stößt man manchmal auf weitere Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher

Zeit bedarf. Daher ist es sinnvoll, beide einzureichende Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand der jeweiligen Aufgabe unangenehm kurz vor Ablauf der Bearbeitungszeit überrascht zu werden und keine vollständige Lösung mehr zu schaffen.

Dokumentation

Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und deren Umsetzung in Software fließen lassen. Doch ohne eine verständliche Beschreibung der Lösungsideen und ihrer jeweiligen Umsetzung, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine geeignete Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (welche die verschiedenen, bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung nur wenig wert.

Bewerterinnen und Bewerter können die Qualität Ihrer Aufgabenlösungen nur anhand dieser Informationen vernünftig einschätzen. Mängel in der Dokumentation der Einsendung können nur selten durch das konkrete Überprüfen der Ergebnisse der Programme durch die Bewerterin oder den Bewerter etwas ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es gelegentlich Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner hinsichtlich Lauffähigkeit getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung hin, in welcher Programmiersprache auch immer. Einige unterhaltsame Formulierungsperlen sind im Anhang wiedergegeben.

Bewertungskriterien

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich nicht um perfekte Musterlösungen, sondern um sinnvolle Lösungsvorschläge, die nicht die einzigen Lösungswege darstellen, die wir gelten ließen. Wir akzeptieren vielmehr in der Regel alle Ansätze, auch ungewöhnliche, kreative Bearbeitungen, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige der Fußnoten in den folgenden Lösungsvorschlägen verweisen auf weiterführende Fachliteratur für wissenschaftlich besonders Interessierte; ihre Referenzierung und ihr tieferes Verständnis wurden natürlich nicht von den Teilnehmenden erwartet.

Unabhängig vom gewählten Lösungsweg gibt es einige Dinge, die auf jeden Fall in der Dokumentation erwartet wurden oder sogar Pluspunkte gaben. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt jeweils am Ende des Lösungsvorschlags erläutert, worauf unter anderem bei der Bewertung dieser Aufgabe besonders geachtet wurde. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation (klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele und wesentliche Auszüge aus dem Quellcode) einschließlich einer theoretischen Analyse (geeignete Laufzeitüberlegungen bzw. eine Diskussion der Komplexität des Problems) sowie an den Quellcode der implementierten Software (mit übersichtlicher Programmstruktur und verständlicher Kommentierung) und an das lauffähige Programm (ohne Implementierungsfehler). Wünschenswert sind unter anderem auch Hinweise

auf die Grenzen des angewandten Verfahrens sowie sinnvolle Begründungen z. B. für Heuristiken, vorgenommene Vereinfachungen und Näherungen. Geeignete Abbildungen und eigene zusätzliche Beispieldaten können die Erläuterungen in der Dokumentation gut unterstützten. Die erhaltenen Rechenergebnisse für die Beispieldaten (ggf. mit Angaben zur Rechenzeit) sollten leicht nachvollziehbar dargestellt sein (z. B. durch die Ausgabe von Zwischenschritten oder geeigneten Visualisierungen). Eine Untersuchung der Skalierbarkeit des eingesetzten Algorithmus hinsichtlich des Umfangs der Eingabedaten ist oft ebenfalls nützlich.

Danksagung

Alle Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt: Peter Rossmanith, *RWTH Aachen University* (Vorsitzender); Hanno Baehr, *RWE Supply & Trading GmbH, Essen*; Jens Gallenbacher, *JGU Mainz*; Rainer Gemulla, *Universität Mannheim*; Torben Hagerup, *Universität Augsburg*; Christof Hanke, *Berufliches Gymnasium für Informatik, FLB Herford*; Thomas Kesselheim, *Universität Bonn*; Arno Pasternak, *Fritz-Steinhoff-Gesamtschule Hagen, TU Dortmund*; Holger Schlingloff, *Fraunhofer FOKUS, HU Berlin*; Melanie Schmidt, *Universität Köln*; als Gast im Ausschuss: Wolfgang Pohl, *BWINF, Bonn*.

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Niels Mündler und Maximilian Azendorf (Aufgabe 1), Niccolò Rigi-Luperti (Aufgabe 2) und Niko Hastrich (Aufgabe 3). Allen Beteiligten sei für Ihre Mitarbeit hiermit ganz herzlich gedankt.

Aufgabe 1: Stromrallye

1.1 Analyse

In der Aufgabenstellung wird zunächst gefordert, ein Programm zu schreiben, dass auf einem Spielbrett mit Batterien einen Roboter so navigiert, dass am Ende der Ladezustand aller Batterien auf 0 gebracht ist. Die Batterien sind dabei auf einem Raster verteilt, also auf einem quadratischen Spielbrett, das in gleich große Felder unterteilt ist.

Der Roboter kann sich in einem Schritt zu einem (nicht-diagonal) benachbarten Feld bewegen. Jeder Schritt des Roboters reduziert die Ladung der getragenen Batterie um 1, die Ladung kann nicht negativ werden. Die Länge des Weges von einem Feld zu einem anderen wird in der Anzahl der Schritte gemessen, die der Roboter für das Zurücklegen des Weges benötigt. Das Ablaufen eines Weges, also eine Sequenz an Schritten, wird im folgenden als allgemeiner Zug bezeichnet. Betritt der Roboter ein Feld, auf dem eine Batterie liegt (er „besucht“ das Feld bzw. die dort liegende Batterie), so wird die dort liegende Batterie aufgenommen und die zuvor getragene Batterie abgelegt.

Eine Anordnung von Batterien mit gegebenen Ladungen auf dem Spielbrett, sowie die Position des Roboters mit einer Ladung wird im Folgenden auch als *Stromrallye-Situation*, Zustand oder (zu lösendes) *Problem* bezeichnet. Zur Beschreibung eines Problems verwenden wir u.a. die folgende Notation:

- Ein $m \times n$-Brett (auch Spielbrett) ist ein Rechteck der Länge m und Breite n, das in $m \times n$ quadratische Felder unterteilt ist.
- Eine Stromrallye-Situation wird insgesamt mit $S = (m, n, B_S, pos_S, |\cdot|_S, r_S)$ notiert. Eine $m \times n$-Situation ist eine Sitation auf einem $m \times n$-Brett.
- Der Roboter $r_S = ((x, y), b)$ befindet sich in der Situation S auf dem Feld (x, y) und trägt Batterie b.
- B_S ist die Menge aller in der Situation S auf dem Spielbrett befindlichen Batterien (einschließlich der Batterie, die der Roboter trägt).
- $|b|_S$ ist die Ladung der Batterie b in Situation S.
- $pos_S(b)$ ist die aktuelle Position der Batterie b in Situation S, bezeichnet also ein Feld. $pos_S(b) = (x, y)$ für die Batterie, die der Roboter trägt.
- $d_S(b_1, b_2)$ ist die Distanz von Batterie b_1 nach Batterie b_2 in Situation S. Diese muss nicht immer der Manhattan-Distanz von $pos_S(b_1)$ und $pos_S(b_2)$ entsprechen, da der Roboter auf dem Weg zwischen zwei Batterien eventuell andere Batterien umgehen muss.

1.1.1 Laufzeitkomplexität

Das Problem ist vermutlich NP-vollständig. Die Vermutung beruht auf der großen Ähnlichkeit zum Problem von Hamiltonpfaden in Gittergraphen, welches NP-vollständig ist. Die Ähnlichkeit ergibt sich besonders, wenn nur Batterien der Ladung 1 betrachtet werden, also jede Batterie genau einmal besucht werden muss und die Bewegung auf Bewegungen zu direkt benachbarten Felder beschränkt sind. Ein mögliches Reduktionsschema ist im folgenden beschrieben.

Grobe Reduktion: Sei $G_{m\times n} = (V,E)$ mit $V = \{v_{i,j}\}$ ein Gittergraph und $V' \subseteq V$. Dann ist das Finden eines Hamiltonpfades in $G' = G_{m\times n}[V']$ NP-vollständig.[1]

Sei $S = (m,n,B_S,pos_S,|\cdot|_S,r_S)$ eine $m \times n$-Stromrallye-Situation, b_r eine Batterie mit Ladung 1, $r_S := ((x,y),b_r)$ und eine Batterie $b \in B_S$ mit $pos_S(b) = (i,j)$ und $|b|_S = 1$ genau dann wenn $v_{i,j} \in V'$.

S ist lösbar genau dann wenn ein Hamiltonpfad mit Startknoten $v_{x,y}$ in G' existiert.

Damit existiert genau dann ein Hamiltonpfad in G, wenn ein (x,y) existiert, für das $(m,n,B_S,pos_S,|\cdot|_S,((x,y),b_r))$ lösbar ist. Die Laufzeit der Reduktion ist damit mit $\mathcal{O}(m^2n^2)$ polynomiell.

1.2 Lösungsidee

1.2.1 Suche durch den Zustandsraum

Grundsätzlich wird im Folgenden eine Suche durch den Zustandsraum angewandt, in der alle gültigen Züge des Roboters ausprobiert werden.

Für den wesentlichen Teil der Suche betrachten wir lediglich spezielle Züge, die von der aktuellen Position des Roboters zu einem Feld führen, auf dem eine Batterie liegt, und dazwischen kein Feld besuchen, auf dem eine andere Batterie liegt.

Damit werden alle Züge abgedeckt, die zu einer Lösung des Problemes führen. Wenn mindestens eine nicht-leere Batterie in der Situation existiert, muss diese auf dem Weg zur Lösung besucht werden. Daher muss jede Sequenz an Schritten, die ein Teil der Lösung ist, ein Feld besuchen, auf dem eine Batterie liegt. Die Ausnahme bilden die „letzten“ Züge, die nur noch die Batterie entleeren, die der Roboter trägt (also $\forall b \in B_S \setminus \{b_r\} : |b|_S = 0$), weshalb diese gesondert behandelt werden.

Ein spezieller Zug wird vereinfacht notiert als Tupel (s,t,d), wobei s und t das Start- und End-Feld des Zuges und d die zurückgelegte Distanz bezeichnen. Dabei wird nicht immer eine eindeutige Sequenz an Schritten beschrieben. Ein Zug ist gültig, wenn eine Sequenz an gültigen Schritten existiert, die von s zu t führen und Distanz d ablaufen. Ein Übergang zwischen zwei Zuständen S_1 und S_2 ist möglich, wenn es einen gültigen Zug (s,t,d) gibt, sodass die Positionen und Ladungen der Batterien von S_2 sich aus der Bewegung des Roboters gemäß des Zuges (s,t,d) ergeben, wenn in S_1 gestartet wird.

Der Zustand $S_2 = (m,n,B_{S_2},pos_{S_2},|\cdot|_{S_2},r_{S_2})$ ergibt sich dann wie folgt:

[1] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM J. Comput.*, 11(4):676–686, 1982.

- Wir bezeichnen mit b_r und b_t die Batterien, die der Roboter in S_1 trägt bzw. sich in S_1 an Position t befindet.
- $B_{S_2} = B_{S_1}$
- $pos_{S_2}(b) = \begin{cases} pos_{S_1}(b_t) & \text{wenn } b = b_r \\ pos_{S_1}(b) & \text{sonst} \end{cases}$
- $|b|_{S_2} = \begin{cases} |b_r|_{S_1} - d & \text{wenn } b = b_r \\ |b|_{S_1} & \text{sonst} \end{cases}$
- $r_{S_2} = (t, b_t)$

1.2.2 Suchprinzip

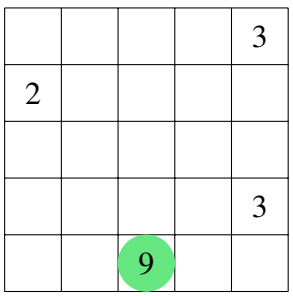
Es wird eine Tiefensuche angewandt, um dem Speicherbedarf der Speicherung aller Zustände bei einer Breitensuche auszuweichen. Da jede Suche erst nach mindestens $|B_S|$ Zügen ein positives Ergebnis liefern kann, wäre der Suchbaum zu diesem Zeitpunkt erwartungsgemäß exponentiell in der Tiefe des Baumes (= Anzahl der Züge) gewachsen. Eine Tiefensuche kann dagegen mit einem Speicherbedarf linear in der Tiefe des Baumes denselben Raum abdecken. Dies wird umgesetzt, indem wir lediglich den aktuellen Zustand speichern sowie die durchgeführten Züge in einem Stack, sodass die umgekehrte Anwendung aller Züge in Stackreihenfolge wieder zum Ausgangszustand führt.

Wenn klar ist, dass aus dem aktuellen Zustand keine Folge weiterer Züge zu einer Lösung führt, wird die Suche hier gestoppt. Dann wird vom vorhergehenden Zustand aus ein neuer Zug gewählt, um einen anderen Folgezustand zu erreichen. Insgesamt gehen wir also nach dem Prinzip des *Backtracking* vor. Um das Durchsuchen von Teilbäumen zu vermeiden, in denen keine Lösung zu finden ist, wird *Pruning* betrieben.

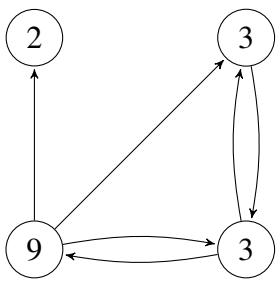
1.2.3 Einschränkungen der Suche

Damit die Suche durch den Zustandsraum nicht zu lange dauert, muss sie sinnvoll eingeschränkt werden. In diesem Abschnitt diskutieren wir dazu einige Möglichkeiten.

1.2.3.1 Bestimmung der Zusammenhangskomponente



Das Feld …



… und der dazugehörige Graph.

Jede Spielsituation S definiert einen gerichteten Erreichbarkeitsgraphen $R_S = (B_S^+, E, d_S)$ mit $B_S^+ = \{b : |b|_S > 0, b \in B_S\}$ und $(b_1, b_2) \in E$ genau dann, wenn $d_S^{\text{Manhattan}}(b_1, b_2) \leq |b_1|_S$[2]. Daraus leiten wir einen „vereinfachten“ ungerichteten Erreichbarkeitsgraphen $T_S = (B_S^+, E', d_S)$ ab mit $E' = \{\{b_1, b_2\} : (b_1, b_2) \in E\}$.

Der vereinfachte Erreichbarkeitsgraph muss zusammenhängend sein[3], sonst ist das Problem nicht lösbar.

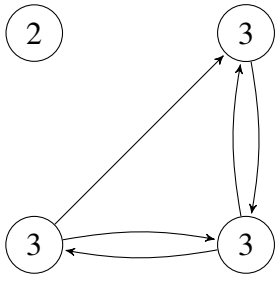
Um das zu begründen, betrachten wir den gerichteten Erreichbarkeitsgraphen. Durch einen Zug können ausgehende Kanten ausschließlich verschwinden, nie entstehen (Dreiecksungleichung). Eingehende Kanten können entstehen, und zwar genau zu der nach einem Zug abgelegten Batterie b_r, da sonst keine Batterie ihre Position verändert und Ladungen nicht größer werden. Die neuen eingehenden Kanten sind genau die eingehenden Kanten zu der aufgenommenen Batterie b_t. b_t ist allerdings bereits in der Zusammenhangskomponente enthalten (sonst könnte b_t nicht besucht werden), und damit auch alle Batterien mit eingehenden Kanten zu b_t. Die Zusammenhangskomponente wächst also nicht. Ist eine Batterie nicht in der Zusammenhangskomponente enthalten, ist es per Definition der Kanten aber auch nicht möglich, die Batterie zu besuchen, denn $d_S(b_1, b_2) \geq d_S^{\text{Manhattan}}(b_1, b_2)$. Da die Zusammenhangskomponente nicht wächst, wird dies auch für alle folgenden Züge nicht möglich sein.

Ein Zug (s, t, d) des Roboters (u, b_r) von S_1 zu S_2 entspricht dem „Entlanggehen“ der Kante (b_r, b_t). Dabei können wir R_{S_2} aus R_{S_1} wie folgt konstruieren:

1. Von der Bordbatterie b_r die Distanz d abziehen, also $|b_r|_{S_2} := |b_r|_{S_1} - d$
2. Die Kanten(-gewichte) im Erreichbarkeitsgraphen gemäß $d_{S_2}^{\text{Manhattan}}$ updaten. Es ändern sich nur die Kanten, die zu b_r inzident sind. Falls $|b_r|_{S_2} > 0$:
   - Entferne Kanten zu Batterien, die nun zu weit entfernt sind.
   - Entferne Kanten von Batterien, die nun zu weit entfernt sind.
   - Übernehme die eingehenden Kanten von b_t.

   Falls $|b_r|_{S_2} = 0$, entferne b_r aus B_S^+ und alle Kanten aus E, die zu b_r inzident sind.

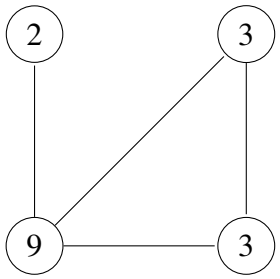
Gehen wir in oben gezeigter Situation zur Batterie ganz oben rechts, ergibt sich



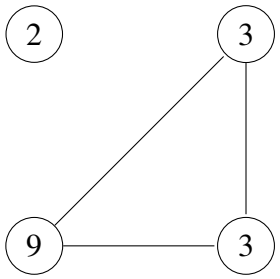
[2] $d_S^{\text{Manhattan}}(b_1, b_2)$ bezeichnet hier im Gegensatz zu $d_S(b_1, b_2)$ einfach die Vektorsumme der absoluten Differenz $|pos_S(b_2) - pos_S(b_1)|$ (Manhattandistanz) und *nicht* die Länge des kürzesten gültigen Weges von b_1 nach b_2.

[3] In einem zusammenhängenden Graphen gibt es von jedem Knoten zu jedem anderen einen Weg.

Der Roboter hat einen Weg der Länge 6 zurückgelegt, die Ladung der Bordbatterie beträgt nun noch 3. Durch den Zug ist die Kante zur Batterie oben links verschwunden, da die Bordbatterie nun nicht mehr genug Ladung hat, um diese Batterie zu erreichen. Der vereinfachte Erreichbarkeitsgraph verändert sich wie folgt:



(vorher)



(nachher)

Um den Zusammenhang von Graphen zu bestimmen gibt es allgemeine Algorithmen ("dynamic decremental connectivity"[4]). Da der vereinfachte Erreichbarkeitsgraph immer zusammenhängend sein muss, vereinfacht sich das Problem aber erheblich.

Der (an Dynamic Connectivity angelehnte) Algorithmus ist dabei wie folgt nach jeder Aktion:

- Starte eine Suche von b_t aus.
- Suche im verbleibenden Graphen alle Batterien, zu denen Kanten entfernt wurden.
- Wird eine Batterie nicht gefunden: Der Graph ist nicht mehr zusammenhängend.

Die abgelegte Batterie b_r wird nur gesucht, falls $|b_r|_{S_2} > 0$. Damit führt automatisch das Wegfallen leerer Batterien nicht zum Verlust des Zusammenhangs.

1.2.3.2 Weglängen

Für jedes Paar von Batterien (a,b) werden zwei Distanzen gespeichert:

- der kürzest mögliche verbindende Pfad m mit Länge $d_S(a,b) = |m|$
- der kürzest mögliche verbindende Pfad d mit Pfadlänge $|d| \geq 3$.

Das liegt daran, dass bei einem Zug von a zu b mit einer Weglänge $|d| \geq 3$ die Strecke um einen beliebigen geraden Wert $2k$ verlängert werden kann, indem in der Mitte des Weges ein Schritt zurück und anschließend wieder nach vorn getan wird. Alternativ kann oft ein Schritt zur Seite und wieder zurück getätigt werden. Kann die Strecke verändert werden, müssen bei der Lösungssuche grundsätzlich alle möglichen Distanzen $|d| + 2k$ $(k \in \mathbb{N}_0)$ betrachtet werden. Falls $|m| \neq |d|$, muss zudem auch noch $|m|$ separat betrachtet werden.

Oftmals können wir uns aber auf lediglich zwei Möglichkeiten beschränken: Entweder der Roboter geht einen kürzesten Weg mit Länge $|m|$ oder einen längsten Weg o mit $|o| = |d| + 2 * \lfloor \frac{|a|_S - |m|}{2} \rfloor$ (die Strecke wird durch die Batterieladung beschränkt).

[4] https://en.wikipedia.org/wiki/Dynamic_connectivity

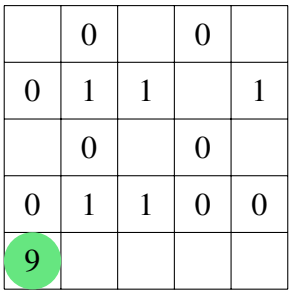


Abbildung 1.1: In dieser Situation existiert keine Lösung, in der die Startbatterie maximal entleert oder minimal entleert wird. Es ist notwendig, mit einer verbleibenden Ladung von genau 4 in den bewegungseinschränkenden Teil des Spielbretts „einzutreten“.

Lemma Existiert eine Lösung für eine Stromrallye-Situation, so existiert fast immer eine Lösung, in der jeder Zug zwischen zwei Batterien einen längsten oder einen kürzesten Weg nimmt.

Begründung Angenommen in einem Zug von Batterie a nach Batterie b mit Mindestdistanz $|m|$ wird ein um $2k$ verlängerter Weg abgelaufen und a wird mit Ladung $l = |a|_S - |m| - 2 * k$ abgelegt. Wir unterscheiden dann zwei Fälle.

1. Im weiteren Verlauf der Lösung wird Batterie a nicht wieder aufgenommen. Dann muss sie in diesem Zug vollständig entladen werden, der Weg muss also ein längster Weg sein.
2. Im weiteren Verlauf der Lösung wird a wieder aufgenommen. Dies kann zwei Gründe haben:
   a) a wird anschließend direkt entladen.

      Anstelle des ursprünglichen Zuges laufen wir mit maximaler Distanz $|m| + 2 * \lfloor \frac{|a|_S - |m|}{2} \rfloor$ (= maximale Entladung) zu b und entladen nach dem erneuten Aufheben von a die verbleibende Ladung $|a|_S - (|m| + 2\lfloor \frac{|a|_S - |m|}{2} \rfloor) = (|a|_S - |m|) \bmod 2$. Da dies stets 0 oder 1 ist, ist ein entsprechender Zug immer möglich. Durch die Abänderung der entsprechenden Züge erhalten wir eine gültige Lösung.

   b) Eine weitere Batterie c wird mit Distanz $i < l$ angelaufen.

      i. c ist mit Distanz $i + 2k$ erreichbar. Dann kann b mit m angelaufen werden und wir erhalten eine gültige Lösung.

      ii. Die Distanz zu c kann nicht zu $i + 2k$ verändert werden (siehe Abbildung 1.1). Diese Optimierung kann dann nicht angewandt werden.

Die Situation, die den nicht optimierbaren Fall in Punkt 2(b)ii hervorruft, kann jedoch erkannt werden. Dazu wird überprüft, ob mit der verbleibenden Ladung l am Zielort eine Batterie c erreicht werden kann, die nicht mit Distanz $|m| + 2$ erreicht werden kann. Dies ist mindestens der Fall, wenn der kürzeste Weg mit Länge ≥ 3 nicht die Länge $|m| + 2$ hat. Dann ist diese Distanz nicht um beliebiges $2k$ veränderbar (konkret für $k = 1$), und beim Zug von a nach b wird jede mögliche Distanz ausprobiert.

1.2.3.3 Unerreichbare Batterien

Hat eine nicht-leere Batterie nur noch ausgehende Kanten, jedoch keine eingehenden Kanten mehr, und es handelt sich nicht um die Batterie die sich aktuell im Roboter befindet, so kann gebacktrackt werden. Der Grund ist, dass diese Batterie nicht mehr erreicht werden kann (siehe Unterunterabschnitt 1.2.3.1). Damit kann die verbleibende Ladung auch nicht mehr entleert werden.

1.2.3.4 Heuristiken bei der Auswahl des nächsten Zuges

Um die Wahrscheinlichkeit zu erhöhen, durch einen Zug nicht in einen Teilbaum zu geraten, der sehr groß ist und mit geringer Wahrscheinlichkeit eine Lösung beinhaltet, können Heuristiken bei der Auswahl des nächsten Zuges angewandt werden.

Eine solche Heuristik muss eine Bewertung möglicher Züge liefern, aus der sich eine Priorisierung der Züge ergibt. Mögliche Einflussfaktoren für eine Bewertung von Aktion (a,b,d) sind:

- die verbleibende Batterieladung $|a|_S - d$ der abgelegten Batterie a (0 ist sehr gut, 1 oft sehr schlecht, höhere Werte dagegen oft akzeptabel)
- die Anzahl r der nach Durchführung von a aus erreichbaren Batterien (ist sie sehr niedrig oder 0, kann a nur durch erneuten Besuch geleert werden)

Konkret wurde in der Beispiellösung folgende Priorisierung verwendet (eine kleinere Zahl bedeutet höhere Priorität):

1. Batterieladung ist 0
2. Batterieladung > 0 und $r > 0$ (Priorisierung untereinander nach r aufsteigend)
3. Batterieladung > 0 und $r = 0$

Bei dieser Priorisierung wird ein kleineres r vorgezogen, da das die möglichen Züge bei Wiederaufnahme einschränkt und der entstehende Teilbaum damit kleiner ist und schneller abgeschlossen wird. Ein r von 0 ist dagegen gleichbedeutend damit, dass die Batterie die als letztes aufzuhebende Batterie sein wird oder von einer Batterie mit sehr hoher Ladung aufgehoben werden muss, was als unwahrscheinlicher eingeschätzt wird.

1.2.4 Weitere Verbesserungen

1.2.4.1 Randomisierung und Parallelisierung der Suche

Da in jeder (Teil-)Situation eine andere Reihenfolge des Besuches benachbarter Batterien erfolgreich sein kann, bringt die beste zu erwartende durchschnittliche Laufzeit eine Randomisierung des Besuches der Nachbarbatterien.[5]

Oft ist zu beobachten, dass sich eine Suche nach einer Zeit „festfährt“, sie befindet sich in einem kleinen Teil des gesamten Spielbretts und probiert in diesem Bereich alle gültigen Züge

[5] Robert Sedgewick. Finding paths in graphs. 2007.

Abbildung 1.2: Beispiel für rechteckig (links) und C-Alphabet (rechts) Teilbereiche von Batterien gleicher Ladung

aus. Bis die Suche eine anfänglich getroffene Entscheidung abändert, muss zunächst der gesamte Teilbaum abgeschlossen werden. Um dieses Problem zu umgehen, wird die Suche nach einem gewissen Zeitintervall abgebrochen und komplett neugestartet. Bei zufälliger Wahl der als nächste zu besuchenden Batterie können schnell frühere Entscheidungen geändert werden und trotzdem durch das Beibehalten der Tiefensuche die Wahrscheinlichkeit einer Lösungsfindung erhöht werden (im Gegensatz zum Ansatz, einfach einen zufälligen Pfad abzulaufen und anschließend vollständig zu verwerfen).

Um dennoch sicherzustellen, dass auch der Fall eines nicht lösbaren Problems erkannt wird, muss parallel zur öfter abbrechenden Suche eine nicht abbrechende Suche durchgeführt werden. Denn nur durch die Überprüfung des gesamten Zustandsraumes kann herausgefunden werden, ob das Problem evtl. nicht lösbar ist. Bei der nicht abbrechenden Suche werden die vorher genannten Heuristiken zur Batteriewahl angewandt anstatt einer völlig zufälligen Auswahl.

Zusätzlich können natürlich viele randomisierte Suchen parallel ablaufen, oder aus jedem Teilbaum ein neuer Task erzeugt werden, der unabhängig bearbeitet wird. Hiermit kann jedoch insgesamt nur ein linearer Speed-Up über dem exponentiell wachsenden Problem erzeugt werden.

1.2.4.2 Zusammenhängende Flächen von Batterien gleicher Ladungen

Für spezielle Formen von Gittergraphen lassen sich in linearer Zeit Hamiltonpfade bzw. deren Existenz bestimmen.[6]

Hierbei handelt es sich um rechteckige Graphen sowie sogenannte Alphabetgraphen (durch ihre Ähnlichkeit zu bestimmten Buchstaben) der Klassen L, C, F und E (siehe Abbildung 1.2). Wenn sich auf dem Spielbrett Batterien in einer solchen Anordnung befinden und alle die gleiche Ladung besitzen, kann dieser Hamiltonpfad dort abgelaufen werden, um sämtliche dieser Batterien um eine Ladungseinheit zu reduzieren.

Die Erkennung sinnvoller (maximaler o.ä.) zusammenhängender Teilbereiche des Spielbretts ist jedoch nicht trivial. Die Bestimmung maximaler, einander nicht überlappender Rechtecke ist aber in einer Laufzeit von $O(n^2)$ machbar, wobei n die Seitenlänge des Spielbretts ist.[7] Jedoch sollte beachtet werden, dass die Batterien nicht statisch sind und damit die Menge der noch vorhandenen Rechtecke stets aktuell gehalten werden muss. Generell sollte die Suche nach

[6] Fatemeh Keshavarz-Kohjerdi and Alireza Bagheri. Hamiltonian paths in some classes of grid graphs. *J. Applied Mathematics*, 2012:475087:1–475087:17, 2012.

[7] David Vandevoorde. The maximal rectangle problem. *Dr. Dobbs Journal*, 1998.

solchen Rechtecken die gesamte zusammenhängende Fläche umfassen, in der sich der Roboter befindet. Für andere Formen sind uns aktuell keine effizienten Algorithmen bekannt.

1.3 Problemschwierigkeit und -erzeugung

1.3.1 Schwierigkeitsmaße

Für die Messung der „Schwierigkeit“ eines Problems bieten sich verschiedene Messgrößen an, die teils direkt, teils statistisch ermittelt werden können und jeweils Nach- und Vorteile haben. Im Folgenden werden mögliche Größen genannt und deren Vor- und Nachteile diskutiert.

Verhältnis erfolgreicher Zugfolgen zu gültigen Zugfolgen Das Problem ist einfach, wenn die Züge des Roboters zufällig gewählt werden und die Wahrscheinlichkeit dabei groß ist, erfolgreich alle Batterien zu entleeren. Letzteres kann verschiedene Gründe haben, zum Beispiel dass ab einem gewissen Punkt nur noch wenige gültige Züge möglich sind, die hohes Erfolgspotential haben, oder dass es viele mögliche Lösungen gibt.

Dieses Maß birgt den Nachteil, dass die Zahl gültiger Zugfolgen typischerweise sehr groß ist. Es bietet sich daher an, den letzteren Wert durch z.B. Monte-Carlo-Methoden statistisch zu ermitteln. Dabei ist ein weiterer Nachteil, dass die Zahl erfolgreicher Zugfolgen typischerweise verhältnismäßig sehr klein ist. Um eine aussagekräftige Metrik (die nicht etwa stets 0 ist) zu erhalten, müssen hier noch weitere Umformungen angewandt werden.

Durchschnittliche Entleerung in gültigen Zugfolgen Statt nur erfolgreiche Zugfolgen zu zählen, kann auch berechnet werden, wie viel Prozent der Ladung im Schnitt in einem Lösungsversuch verbraucht wird. Dies kann eine Metrik sein, die darauf hinweist, wie leicht es ist, einen großen Teil des Problems zu lösen.

Entscheidender Nachteil ist hier, dass das Entleeren eines hohen Prozentsatzes der Batterien nicht notwendigerweise einer Annäherung an die Lösung gleichkommt. Denkbar sind Probleme, in der nur durch eine spezielle Zugfolge 100% der Ladung entleert werden kann, während viele Zugfolgen in eine „Sackgasse“ führen, in der nur eine Batterie übrig bleibt. Auch diese Metrik muss statistisch erhoben werden.

Durchschnittliche Anzahl Alternativen pro Schritt Ein anderer Ansatz ist zu messen, wie stark sich der Suchbaum auffächert, wie viele Optionen also bei der Auswahl des nächsten Zuges möglich sind. Zudem kann es sinnvoll sein in Betracht zu ziehen, welcher Prozentsatz dieser Optionen zu einer Lösung führt. Ein kleines Verhältnis (eine Aktion von hunderten) würde darauf hinweisen, dass es notwendig ist, sehr genau zu wählen.

Schwierig ist jedoch die Ermittlung dieses Wertes, auch hier bietet sich eine statistische Erhebung an. Lediglich die Zahl der Alternativen bei der Zugauswahl zu betrachten kann jedoch sehr irreführen. So sind zum Beispiel in einem gefüllten Quadrat stets nur 4 Aktionen möglich, die richtige Zugfolge zu wählen ist dennoch schwierig. Auch ein Verhältnis kann hier in die Irre führen, da die Länge der Zugfolge hier besonders groß ist und das Problem erschwert.

Anzahl kritischer Entscheidungen Dies führt zu dem Ansatz, die totale Zahl an Entscheidungen zu erfassen, welche kritisch für das erfolgreiche Entleeren aller Batterien sind. Kritisch ist hier ein vager Begriff, der durch ein kleine absolute Zahl an Optionen (1 oder 2) oder ein kleines Verhältnis bestimmt sein kann.

Auch kann man überlegen hier den Zeitpunkt der Entscheidung einfließen zu lassen bzw. die Länge des verbleibenden Pfades. Frühe kritische Entscheidungen sind deutlich schwerwiegender, da möglicherweise ein großer verbleibender Suchbaum erforscht wird, von dem kein einziger Pfad zur Lösung führen wird.

1.3.2 Probleme erzeugen

Zunächst ist wichtig, dass bei der Erzeugung von Problemen sichergestellt wird, dass diese lösbar sind, wie in der Aufgabenstellung gefordert. Einfach zufällig Felder zu erzeugen und im Nachhinein auf Lösbarkeit zu überprüfen ist (ohne sinnvolle Erhöhung der Lösbarkeitsrate) sehr ineffizient. Sinnvoll sind Erzeugungsansätze, die eine Lösung mitliefern.

1.3.2.1 Zufälliges Platzieren und Besuchen von Batterien

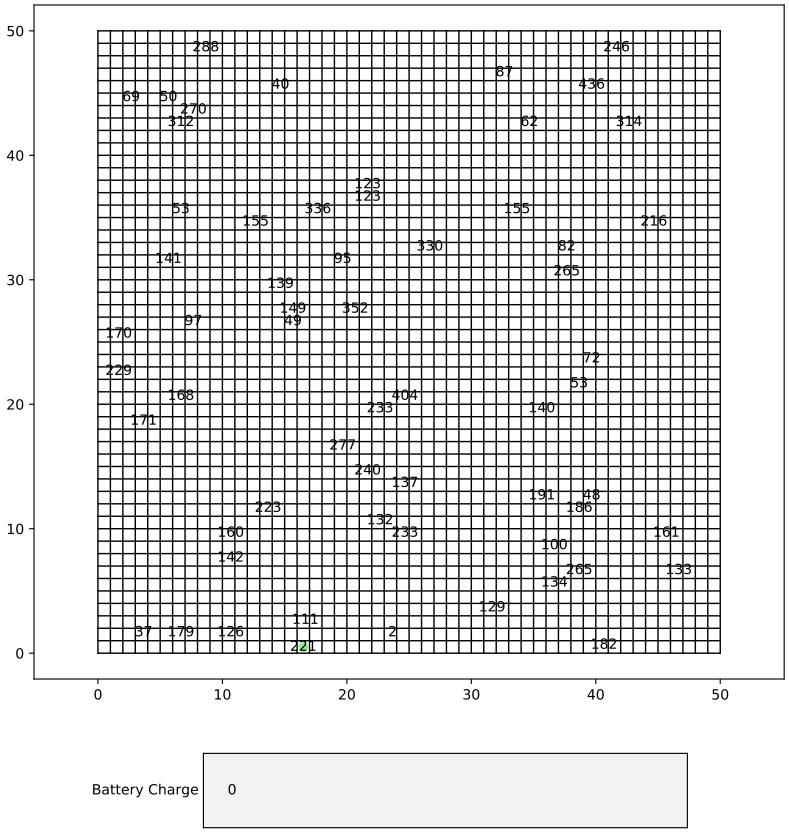
Ein einfacher Ansatz gültige Lösungen inklusive Lösungsweg zu generieren basiert auf zwei Schritten. Zunächst werden Batterien auf zufälligen Positionen auf dem Feld verteilt und erhalten Ladung 0. Anschließend zieht ein virtueller Roboter in zufälliger Folge über die Batterien, woraus schließlich die Lösungs-Zugfolge wird. Dazu wird das Spielbrett wie folgt abgelaufen:

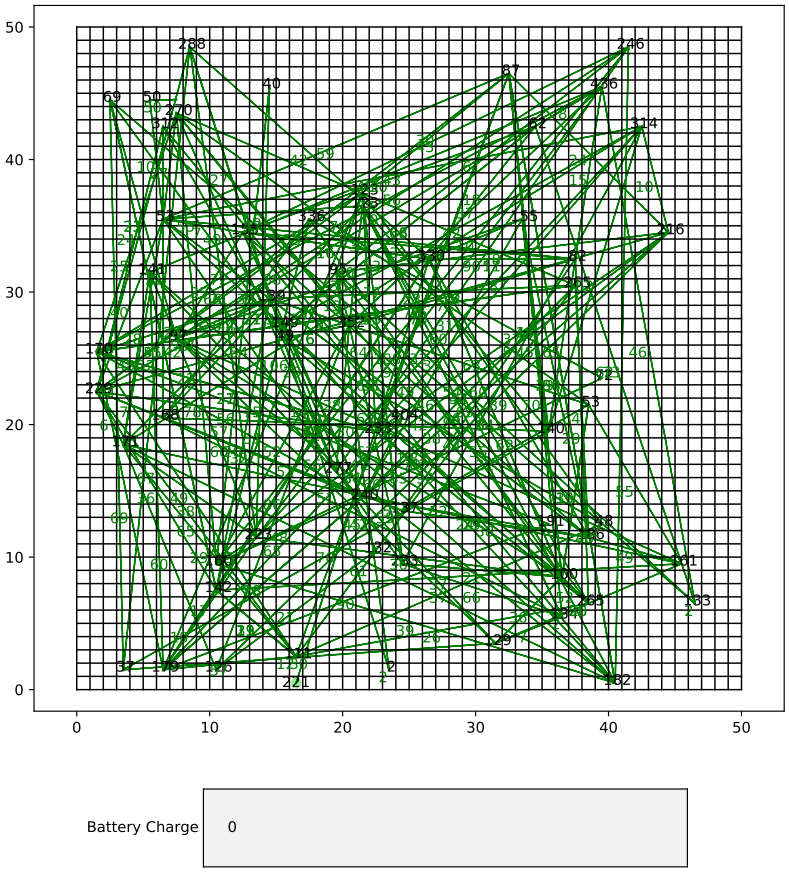
Der virtuelle Roboter startet mit Batterie a und besucht zufällig eine andere Batterie b auf dem Feld in Distanz d. Dabei wird nach dem Besuch der Batterie b der verwendeten Batterie a eine Ladung von $d + 2 * k$ ($k \in \mathbb{N}_0$ beliebig/zufällig) hinzugefügt. Die Batterien werden nun wie im Spiel ausgetauscht und der Roboter beginnt mit dem nächsten Lauf.

Dies wird vorzugsweise solange wiederholt, bis alle Batterien besucht wurden (da die unbenutzten Batterien eine Ladung von 0 haben, ist es aber nicht notwendig). Die entstehende Zugfolge des Roboters bildet eine gültige Lösung für das Problem, auf dem alle Batterien an ihrer Ursprungsposition sind und mit der erzeugten Ladung versehen sind.

Die resultierenden Felder weisen typischerweise eine hohe Ladung an jeder Batterie auf, da, bis genügend Batterien besucht wurden, einige Batterien mehrfach verwendet wurden (siehe Abbildung 1.3). Das kann manchmal dazu führen, dass eine deutlich einfachere Lösung möglich wird. Um dies zu verhindern kann zum Beispiel erzwungen werden, dass noch nicht besuchte Batterien mit einer höheren Wahrscheinlichkeit aufgesucht werden als bereits besuchte Batterien.

Typischerweise sorgen die hohen Ladungen jedoch dafür, dass in jedem Schritt viele Alternativen offen stehen und der Suchbaum lange breit auffächert. In der Metrik der Alternativen pro Zug würde das Verfahren also gut abschneiden. Die tatsächliche Anzahl kritischer Entscheidungen ist jedoch erwartungsgemäß relativ niedrig, da schwerwiegende Entscheidungen bei Zügen selten und wenn, dann nur zufällig, erzwungen werden.


Battery Charge
0


Battery Charge
0

1.3.2.2 Verwenden von Baublöcken

Um größere Stücke des Problems schnell zu generieren und zum Beispiel älteres Wissen zu verwenden oder Parallelisierung zu ermöglichen, könnten gelöste Teilprobleme zusammen gesetzt werden. Hierbei wäre auch sinnvoll auf spezielle Bausteine (z.B. Rechtecke oder F-/E-/L-Stücke) zurückzugreifen. Die Schwierigkeit wäre dann, diese Unterprobleme zu erkennen und in der richtigen Reihenfolge zu durchlaufen.

Durch geschicktes Einfügen der Stücke auf ein Feld kann reguliert werden, wie viele Pfade zwischen den Stücken existieren. Da sämtliche Bausteine für eine Lösung durchlaufen werden müssen, wird damit eine gewisse Durchlaufreihenfolge erzwungen, die die Anzahl kritischer Entscheidungen erhöht.

1.4 Beispiele

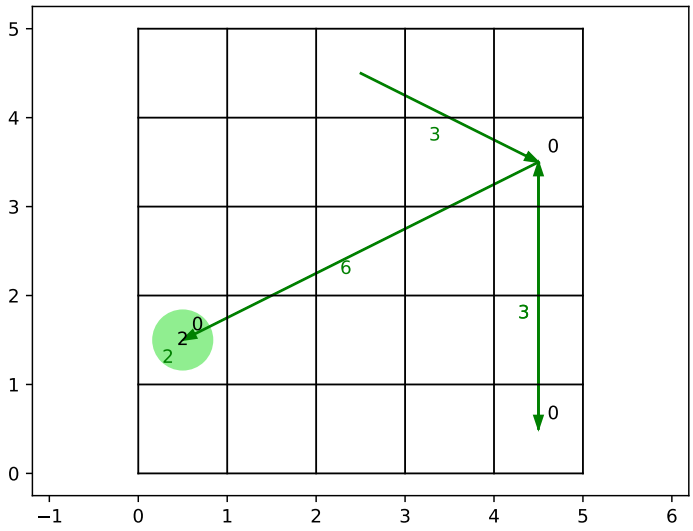
Das implementierte Programm hat in diesem Fall lediglich die Zugfolge abgespeichert und keine detaillierte Schrittsequenz erzeugt. Die Schrittsequenz ist jedoch stets implizit abgelaufen worden und könnte recht simpel (jedoch durch reine Fleißarbeit) erzeugt werden. Daher werden im folgenden nur die abstrakten Züge von einer Batterie zur nächsten gezeigt. Die dabei zurückgelegte Distanz ist in Grün neben dem Pfeil eingezeichnet, der die Positionsveränderung anzeigt.[8]

Für gewöhnlich werden Lösungen von der randomisierten Suche gefunden; deren Ergebnisse sind dann angegeben. Werden bei einem Beispiel zwei Ausführungszeiten dokumentiert, hat die nicht abbrechende Suche zuerst eine Lösung gefunden, welche ebenfalls ausgegeben wurde.

[8] Überlappungen der Texte konnten leider nicht vermieden werden.

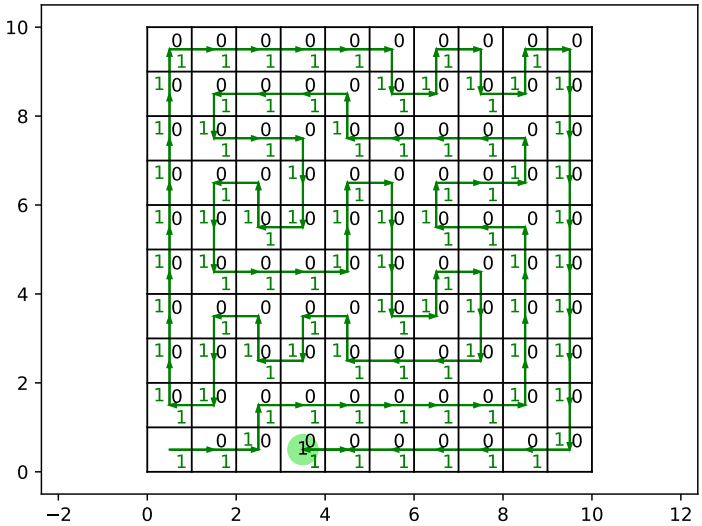
1.4.1 Ergebnisse für vorgegebene und andere Probleme

1.4.1.1 stromrallye0.txt



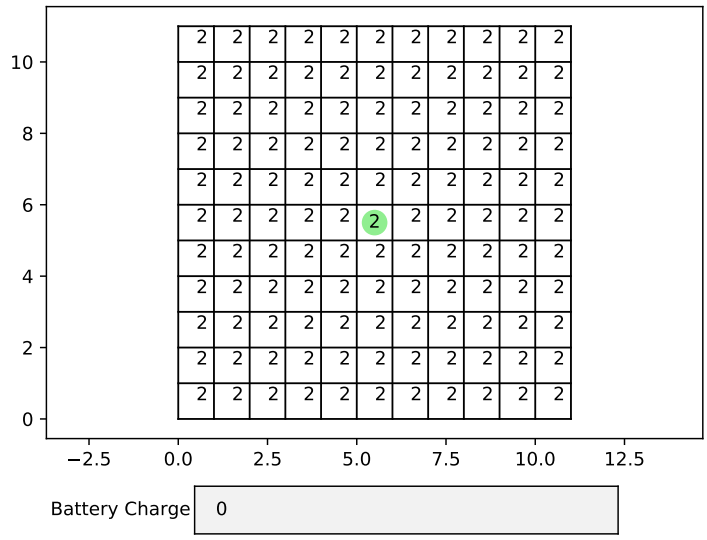

```
deque([((3, 5), (5, 4), 3), ((5, 4), (5, 1), 3),((5, 1), (5, 4), 3),
       ((5, 4), (1, 2), 6), ((1, 2), (1, 2), 2)])
Took 0.005772590637207031 seconds
```

1.4.1.2 stromrallye1.txt



```
deque([((1, 1), (2, 1), 1), ((2, 1), (3, 1), 1), ((3, 1), (4, 1), 1),
       ((4, 1), (5, 1), 1), ((5, 1), (6, 1), 1), ((6, 1), (7, 1), 1),
       ((7, 1), (8, 1), 1), ((8, 1), (9, 1), 1), ((9, 1), (10, 1), 1), ...
Took 0.04112863540649414 seconds
deque([((1, 1), (2, 1), 1), ((2, 1), (3, 1), 1), ((3, 1), (3, 2), 1),
       ((3, 2), (4, 2), 1), ((4, 2), (5, 2), 1), ((5, 2), (6, 2), 1),
       ((6, 2), (7, 2), 1), ((7, 2), (8, 2), 1), ((8, 2), (9, 2), 1), ...
Took 0.24745917320251465 seconds
```

1.4.1.3 stromrallye2.txt



Dieses Problem kann so gelöst werden:

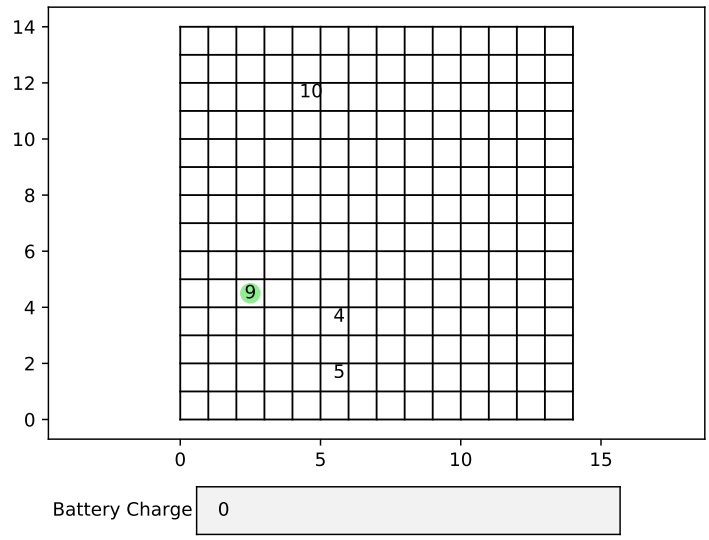
1. Spiralförmig je einen Schritt nach außen laufen.

2. Einen schritt nach innen und wieder zurück gehen, um die äußerste Batterie zu entladen (nun liegt überall 1 und der Roboter trägt 1).

3. Anschließend spiralförmig je einen Schritt nach innen laufen.

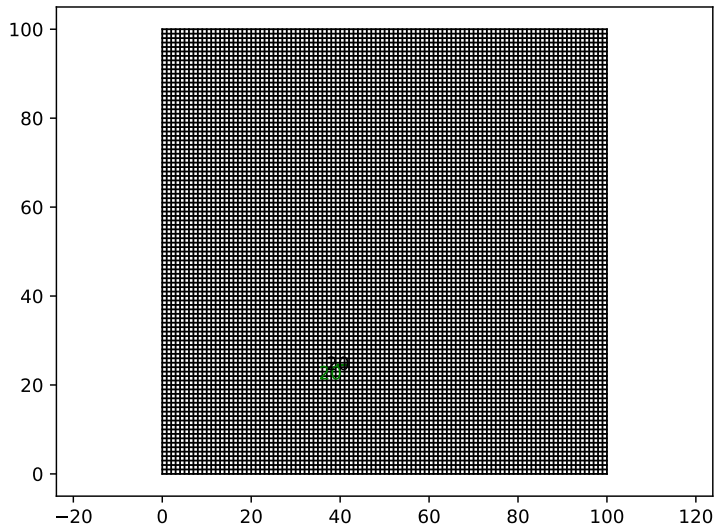
Durch die schiere Anzahl an Batterien und möglichen Schrittfolgen (ohne dass der Graph schnell seinen Zusammenhang verliert oder offensichtlich unlösbar wird), kann dieses Problem vermutlich nur mit dem in Unterunterabschnitt 1.2.4.2 beschriebenen speziellen Lösungsfinder gelöst werden. Dabei muss beachtet werden, dass der Graph durch die in der Mitte fehlende Batterie nach dem Heraustreten in mehrere Rechtecke zerfällt. Entweder diese werden einzeln erkannt oder es wird der Sonderfall beachtet, dass, wenn das Rechteck die letzten Batterien umfasst und innerhalb des Rechteckes ein Loch ist, dieses begangen werden kann.

1.4.1.4 stromrallye3.txt



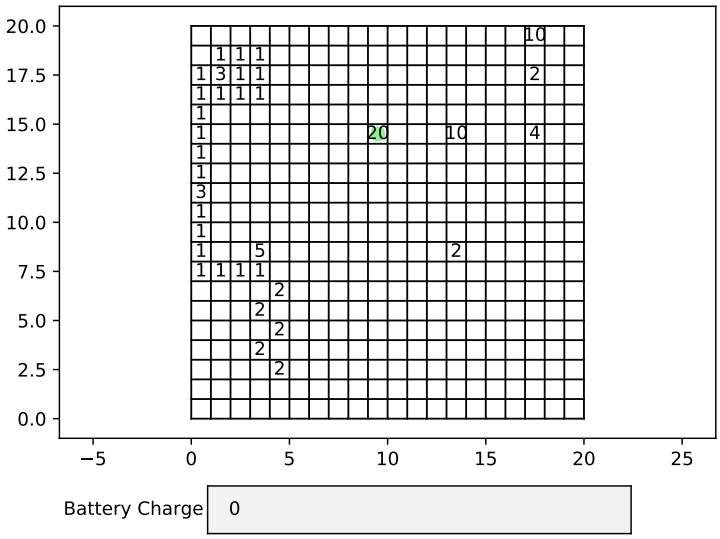

```
Search exhausted, nothing found!
Took 0.023613691329956055 seconds
```

1.4.1.5 stromrallye4.txt




```
deque([((40, 25), (42, 25), 20)])
Took 0.0586698055267334
```

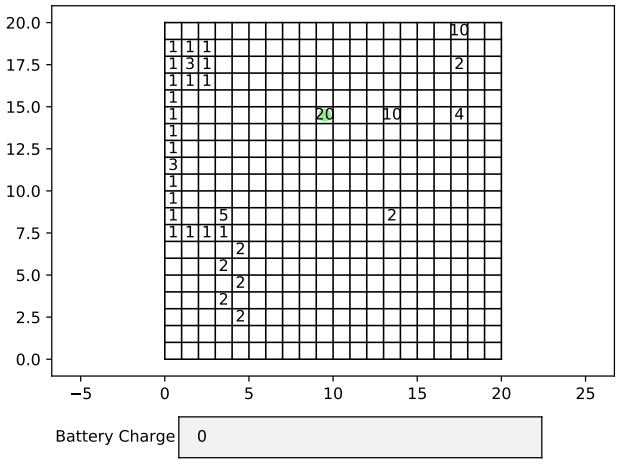
1.4.1.6 stromrallye5.txt

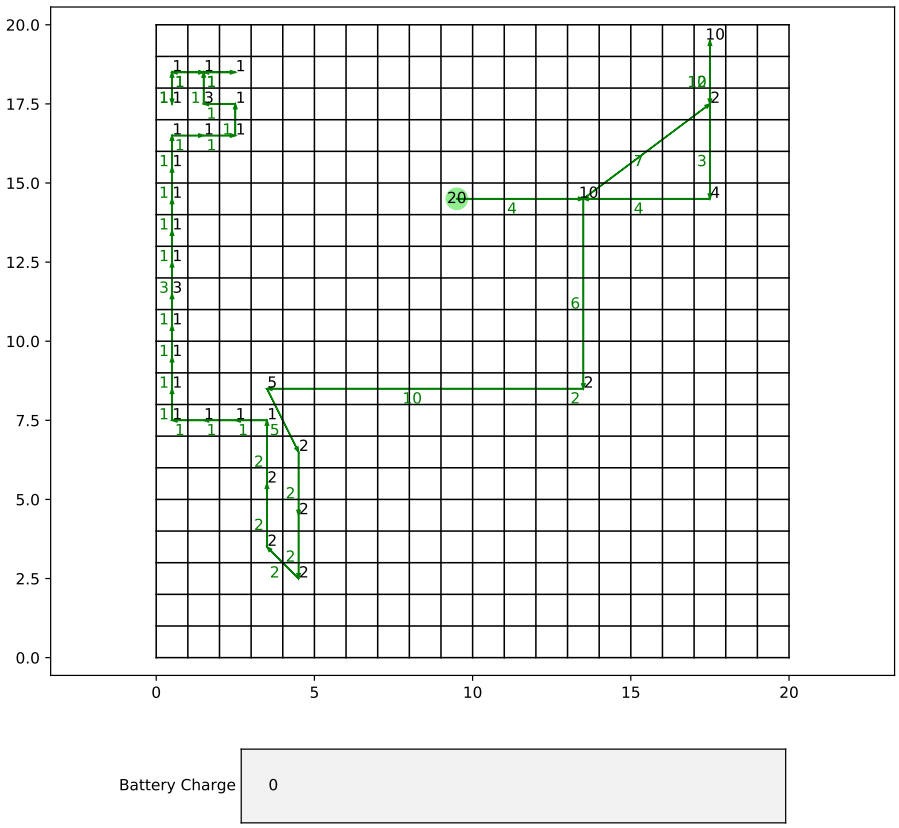



```
Search exhausted, nothing found!
Took 12.235079288482666 seconds
```

1.4.1.7 feld7

Hierbei handelt es sich um eine abgewandelte Version von Stromrallye 5, die lösbar ist. Das Ergebnis wurde von der nicht abbrechenden Suche zuerst gefunden.

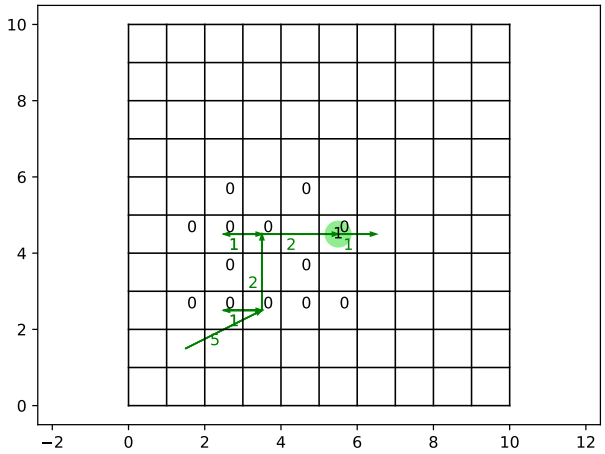





```
deque([((10, 15), (14, 15), 4), ((14, 15), (18, 18), 7), ((18, 18), (18, 20), 2),
       ((18, 20), (18, 18), 10), ((18, 18), (18, 15), 3), ((18, 15), (14, 15), 4),
       ((14, 15), (14, 9), 6), ((14, 9), (14, 9), 2), ((14, 9), (4, 9), 10), ...
Took 2.1835951805114746 seconds
```

1.4.1.8 feld12

In diesem Beispiel wird der Sonderfall behandelt, dass bei nur längster oder nur kürzester Weglänge nicht immer eine Lösung gefunden wird (siehe Abbildung 1.1).

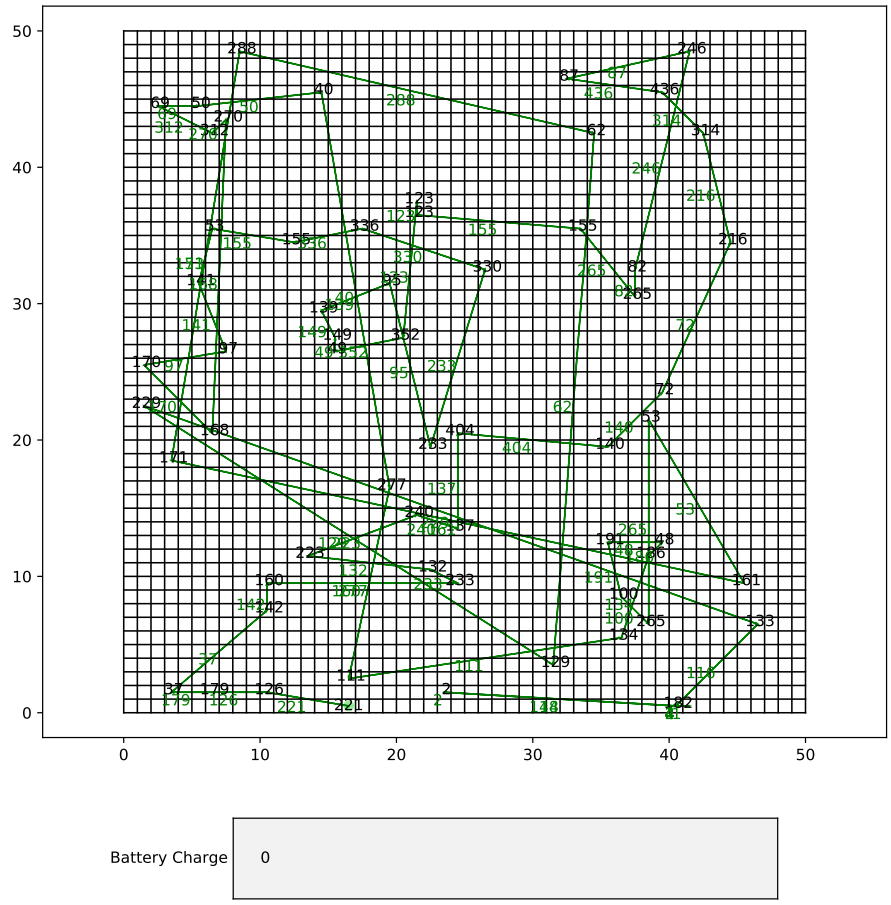


```
deque([((2, 2), (4, 3), 5), ((4, 3), (3, 3), 1), ((3, 3), (4, 3), 1),
       ((4, 3), (4, 5), 2), ((4, 5), (3, 5), 1), ((3, 5), (4, 5), 1),
       ((4, 5), (6, 5), 2), ((6, 5), (7, 5), 1)])
Took 0.008321046829223633 seconds
```

1.4.2 Erzeugte Probleme

1.4.2.1 feld50

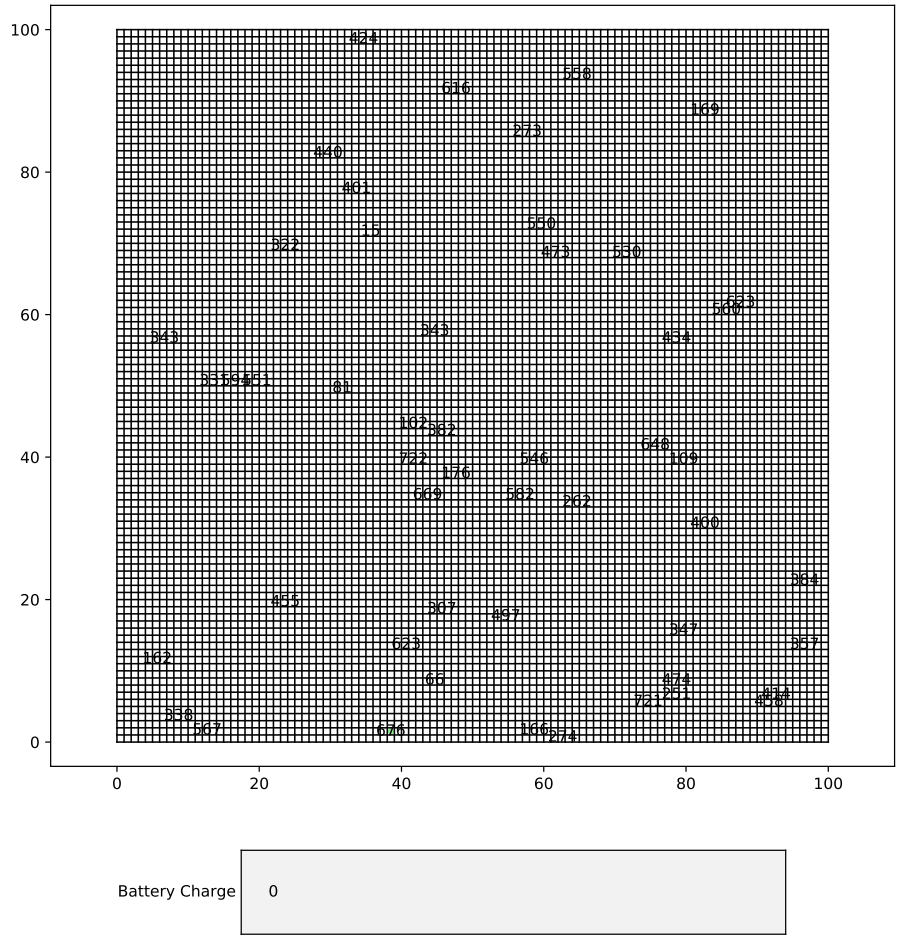
Dieses Beispiel wurde wie in Unterunterabschnitt 1.3.2.1 beschrieben erzeugt, was sich durch die großen Batterieladungen zeigt. Die generierte Lösung aus Abbildung 1.3 ist deutlich verworrener als die hier gezeigte, von der (systematischen) Suche gefundene Lösung.




```
deque([((17, 1), (11, 2), 221), ((11, 2), (7, 2), 126), ((7, 2), (4, 2), 179),
       ((4, 2), (11, 8), 37), ((11, 8), (11, 10), 142), ((11, 10), (25, 10), 160),
       ((25, 10), (23, 11), 233), ((23, 11), ...
Took 3.860287666320801 seconds
```

1.4.2.2 feld100

Dieses Beispiel wurde wie in Unterunterabschnitt 1.3.2.1 beschrieben erzeugt, was sich durch die großen Batterieladungen zeigt. Das Lösungsprogramm konnte hierzu in sinnvoller Zeit keine Lösung finden, obwohl die Anzahl der Batterien geringer ist als in Unterunterabschnitt 1.4.2.1.



1.5 Alternative Verfahren und mögliche Erweiterungen

Insgesamt ist es denkbar, die Heuristiken zur Aktionsauswahl durch theoretische und empirische Analyse zu verbessern oder ganz einem durch Reinforcement Learning trainierten Neuronalen Netzwerk zu überlassen. Wichtig ist hierbei eine sinnvolle Begründung für das gute Funktionieren des erzeugten Systems.

Zur Reduzierung des Speicheraufwands kann statt der Tiefensuche eine Breitensuche zusammen mit einer sehr effizienten Zustandsspeicherung (z.B. Bitmaps) eingesetzt werden.

Mögliche Erweiterungen umfassen:

- Ein interaktives System, um an bestimmten Stellen eine Entscheidungshilfe zu liefern.
- Anders geformte Felder, Wände oder andere Hindernisse (wobei Wände äquivalent zur Positionierung von 0-Batterien sind und daher allein keine Bonuspunkte erhalten, ihre geschickte Verwendung in der Problemerzeugung dagegen schon).

- Felder oder Hindernisse, die mehr oder weniger Ladung verbrauchen wenn sie besucht werden.
- Mehrere Roboter, die parallel gesteuert werden können.
- Reelle Ladungen und eine kontinuierliche Ebene.
- Einführen weiterer Dimensionen.

1.6 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*
- Es liegt nahe, die Erreichbarkeitsbeziehung zwischen den Batterien als Graph zu modellieren. Nötig ist das aber nicht, eine implizite Modellierung der Erreichbarkeit bei der Suche genügt.
- Die Positionen der Batterien ändern sich während der Suche nicht (mit Ausnahme der Bordbatterie). Die Entfernungen zwischen Batterie-Paaren können also vorberechnet werden. Pluspunkt!
- Der kürzeste Weg zwischen zwei Batterien ist sehr häufig nicht die Manhattandistanz, da Batterien im Weg liegen können. Wenn das übersehen wurde, werden Punkte abgezogen (bis -2).
- Für Züge zwischen zwei Batterien müssen auch Umwege mit bedacht werden (ansonsten Abzug bis -2). Diese können für das Leeren von Batterien nützlich sein. Es ist in Ordnung, sich auf den kürzesten Weg (mit Umgehung anderer Batterien) und den längsten Weg (bei der Zielankunft ist die Bordbatterie leer) zu beschränken.

(2) *Teil a: Verfahren allgemein anwendbar*
- Auch wenn einige Beispiele spezielle Lösungsverfahren provozieren, sollte ein auf beliebige Probleme anwendbares Verfahren realisiert worden sein.
- Wurden zusätzlich Verfahren realisiert, die bestimmte Problemkonstellationen gezielt lösen können, gibt es Pluspunkte.

(3) *Teil a: Laufzeit des Verfahrens in Ordnung*
- Die Lösung von Stromrallye-Problemen ist im Allgemeinen schwierig. Mehr als eine geschickte Lösungssuche kann also nicht verlangt werden. Aber der Suchraum lässt sich, wie beschrieben, auf verschiedene Weisen einschränken (Pruning), so dass immer wieder ein Backtracking möglich wird. Es wird erwartet, dass Ideen zur Einschränkung des Suchraums angesprochen und umgesetzt sind.
- Das Verfahren sollte alle vorgegebenen Beispiele in akzeptabler Zeit lösen können (mit Ausnahme von Feld 2) bzw. feststellen können, dass sie nicht lösbar sind.

(4) *Teil a: Speicherbedarf in Ordnung*: Das Verfahren sollte nicht durch unnötig hohen Speicherbedarf ausgebremst werden, was etwa bei einer Breitensuche leicht passieren kann.

(5) *Teil a: Lösbarkeit korrekt entschieden*: Lösungen sollten nur für solche Probleme gefunden werden, die auch wirklich lösbar sind. Falls ein Problem nicht lösbar ist, muss das Verfahren dies herausfinden können.

(6) *Teil b: Schwierigkeitsmaß(e) sinnvoll definiert*

- Es muss mindestens ein Schwierigkeitsmaß angegeben werden. Große Spielbretter mit vielen Batterien sind immer schwierig. Die Schwierigkeit erhöht sich auch, wenn Batteriepositionen mehrfach besucht werden müssen. Größe und Batteriezahl dürfen begrenzt werden, da Probleme für menschliche Spieler erzeugt werden sollen.
- Aus der Beschreibung oder Definition eines Maßes muss klar hervorgehen, dass es automatisch berechnet werden kann.
- Wenn Schwierigkeitsmaße besonders gezielt auf menschliche Spieler abgestimmt sind, kann das mit Pluspunkten belohnt werden.

(7) *Teil b: Lösbare Probleme erzeugt*
- Es muss sichergestellt werden, dass das gewählte Verfahren nur lösbare Probleme erzeugt.
- Das Miterzeugen einer gültigen Lösung ist sinnvoll und hilfreich, aber nicht notwendig.
- Wenn Schwierigkeitswerte explizit (also als Eingabeparameter) in die Erzeugung einfließen, gibt das 1-2 Pluspunkte.

2. Theoretische Analyse

(1) *Verfahren / Qualität ingesamt gut begründet*: Speziell in Teil b muss begründet sein (oder sich offensichtlich aus der Beschreibung des Verfahrens ergeben), dass nur lösbare Probleme generiert werden.

(2) *Gute Überlegungen zur Laufzeit*: Da in jedem Schritt eine Vielzahl Aktionen möglich ist und die genaue Anzahl wie die Tiefe des Baumes variabel ist, ist eine genaue Laufzeitabschätzung sehr schwierig. Der Baum kann allerdings maximal $\sum_{b \in B_F} |b|$ tief sein (wenn man jeden Schritt einzeln betrachtet, er hat dann eine Fächerung zwischen 1 und 4). Derartige Abschätzungen sind also möglich.

(3) *Teil b: Schwierigkeitsmaße diskutiert*: Es sollte (wenigstens kurz) darauf eingegangen werden, warum anhand eines Maßes hoch bewertete Probleme schwierig sind.

3. Dokumentation

(3) *Teil a: Vorgegebene Beispiele dokumentiert*: Es sollte zu allen Beispielen (1 bis 5) angegeben werden, wie sie gelöst bzw. dass sie nicht gelöst werden können. Es ist akzeptabel, wenn Beispiel 0 (aus der Aufgabenstellung) ausgelassen wird.

(5) *Teil b: Erzeugte Probleme dokumentiert*: Mindestens zwei erzeugte Probleme sollten angegeben sein, inklusive einer Lösung.

(6) *Ergebnisse nachvollziehbar dargestellt*: Die grafische Darstellung von Lösungen ist naheliegend und wünschenswert. Gefordert wird das aber nicht. Einigermaßen übersichtlich sollte die Darstellung aber sein. Die Züge des Roboters sollten gut nachvollzogen werden können. Eine einfache Auflistung aller bei der Bewegung über das Feld besuchten Koordinaten (also auch der Felder, auf denen sich keine Batterie befindet) ist nicht ausreichend.

Aufgabe 2: Geburtstag

2.1 Lösungsidee

Diese Aufgabe kann folgendermaßen gelöst werden: Man kombiniert die Startziffer durch mathematische Operationen wiederholt mit sich selbst. Dadurch entstehen schrittweise längere Terme und somit neue Zahlen, welche wiederum miteinander kombiniert werden können zu neuen Zahlen. Dieses Verfahren lässt man fortlaufen bis eine der neuen entdeckten Zahlen die Zielzahl ist. Der Term für die Zielzahl kann dann aus dem gespeicherten Rechnungsverlauf ausgelesen werden.

Start:	**4**
Terme mit 2 Ziffern:	$4+4=8$
	$4\cdot 4=16$
	$4/4=1$
Terme mit 3 Ziffern:	$4+8=12$
	$4+16=20$
	$4+1=5$
	...
Terme mit 4 Ziffern:	**...**

Abbildung 2.1: Grundidee: Neue Zahlen werden durch Kombination von schon bekannten Zahlen generiert. Man geht dabei aufsteigend nach der Anzahl an verwendeten Ziffern vor.

Die Bedingung, dass der gefundene Term möglichst wenige Ziffern erhalten soll, ist eine zentrale Bedingung und kann folgendermaßen erreicht werden: Man berechnet *alle* mathematisch erlaubten Terme, aufsteigend in ihrer Anzahl an Ziffern, und wertet ihr jeweiliges Ergebnis, ihre Zahl, aus. Man generiert erst *alle* Zahlen die sich mit 2 Ziffern darstellen lassen, dann *alle* Zahlen die sich mit 3 Ziffern darstellen lassen, usw., siehe auch Abb. 2.1.

Diese Reihenfolge stellt sicher, dass jede generierte Zahl durch eine minimale Anzahl an Ziffern dargestellt wird. Stößt man bei n zugelassenen Ziffern das erste Mal auf eine bisher unentdeckte Zahl, dann weiß man, dass diese nicht mit $n-1$ oder weniger Ziffern darzustellen ist, denn ansonsten wäre man bereits früher auf sie gestoßen. Die Tatsache dass man *alle* mathematisch zulässigen Zahlen generiert bedeutet dass man unter Umständen sehr viele (hunderttausende) Zwischenergebnisse erhält, bevor man auf die Zielzahl stößt. Mit hinreichender Implementierung lässt sich dies jedoch für die Beispieleingaben in wenigen Sekunden erreichen.

Im Folgenden zeigen wir, wie sich dieses Verfahren praktisch umsetzen lässt, insbesondere in Hinblick auf endlichen Speicherplatz und endliche Zeit. Ein besonderes Augenmerk werden wir auf die „Geteilt“-Operation legen, denn sie führt zu nicht-ganzzahligen Zwischenergebnissen. Diese sind aufwändiger zu handhaben als nur ganzzahlige Zwischenergebnisse, erlauben aber das Entdecken von noch kürzeren Termen, verglichen mit Implementierungen, die sich nur auf ganzzahlige Zwischenergebnisse beschränken.

2.2 Umsetzung

Zentraler Bestandteil unseres Vorgehens sind die *Kosten* einer jeden Zahl. Damit meinen wir die Anzahl an Startziffern die nötig sind um eine Zahl darzustellen. Solche Kosten sind beispielhaft in Tabelle 2.1 gezeigt.

Zahl	zugeh. Term	*Kosten*
4	4	1
16	4*4	2
5	4+4/4	3
3	4-4/4	3

Tabelle 2.1: Zuordnung von Kosten zu jeder Zahl.

Als Datenstruktur führen wir Listen $K[i]$ ein, $i \geqslant 1$. Sie sind zunächst leer. In diesen Listen werden die neu generierten Zahlen gespeichert, aufgeteilt nach ihren Kosten: Alle Zahlen mit Kosten i werden in Liste $K[i]$ gespeichert. Die Startziffer wird zu Liste $K[1]$ hinzugefügt, und, falls Teilaufgabe b) bearbeitet wird, auch die Fakultät der Startziffer. Beide sind die einzigen Zahlen mit Kosten 1. Abb. 2.2 zeigt beispielhaft die Liste $K[1]$ für Startziffer 4.

$$\mathrm{K}[1] = \{4, 24\}$$

Abbildung 2.2: Die Liste $K[1]$: alle Zahlen mit Kosten 1.

Wir wollen nun die weiteren Listen $K[i], i \geqslant 2$ berechnen. Das machen wir aufsteigend in den Kosten, d.h. wir berechnen erst die Liste $K[2]$, dann die Liste $K[3]$, und so weiter. Dadurch wird jede neu entdeckte Zahl durch einen günstigst-möglichen Term generiert. Das macht man sich schnell klar, wenn man z.B. die Situation betrachtet, bei der die Liste $K[5]$ berechnet wird. Stößt man dort das erste Mal auf eine neue Zahl x, dann weiß man, dass es keine günstigere Darstellung mit Kosten 4 oder weniger für x gibt, den ansonsten wäre man dieser Darstellung bereits in $K[4]$ oder noch früher begegnet. Diese Argumentation ist für alle generierten Zahlen gültig, d.h. insbesondere auch die Zielzahl. Wird sie das erste Mal generiert, dann gemäß dieser Vorgehensweise durch einen günstigst-möglichen Term.

Neue Zahlen werden generiert durch Kombinieren von zwei bereits bekannten Zahlen mittels einer mathematischen Operation. Zwei bekannte Zahlen $a \in K[i]$ und $b \in K[j]$ werden benutzt um via $c := a \oplus b$ eine neue Zahl c zu berechnen, mit den Operationen $\oplus \in \{+, -, *, /, \hat{}\}$. Um

Notation zu sparen meinen wir mit $a \oplus b$ auch immer gleichzeitig das Berechnen der vertauschten Variante $b \oplus a$, wegen der Asymmetrie der Operationen $\{-, /, \hat{}\}$. Jede neu generierte Zahl c hat Kosten $k := i + j$, gerade die Summe der Kosten der zugrundeliegenden Zahlen a und b. Die neue Zahl c wird folglich in der Liste $K[i+j]$ gespeichert. Falls c ganzzahlig, positiv und nicht zu groß ist wird auch die Fakultät angewendet und $c!$ in $K[i+j]$ gespeichert. Solche gespeicherten Zahlen nennen wir in den folgenden Seiten auch „gefunden“, „generiert“ oder „entdeckt“, immer mit der Motivation, dass mit den Zahlen gearbeitet werden kann, sobald sie gespeichert wurden. Wir werden uns im Folgenden mit Teilaufgabe b) und damit dem allgemeineren Fall befassen. Für Teilaufgabe a) lässt man einfach Potenz und Fakultät weg, vom Ablauf her ändert sich nichts.

$$
\begin{array}{lllll}
K[1] = \{4, 4!\} & & & & \\
K[2] = K[1] \oplus K[1] & & & & \\
K[3] = K[1] \oplus K[2] & & & & \\
K[4] = K[1] \oplus K[3] & + & K[2] \oplus K[2] & & \\
K[5] = K[1] \oplus K[4] & + & K[2] \oplus K[3] & & \\
K[6] = K[1] \oplus K[5] & + & K[2] \oplus K[4] & + & K[3] \oplus K[3] \\
\ldots & & & &
\end{array}
$$

Abbildung 2.3: Berechnung neuer Listen aus ihren Vorgängern.

Für anvisierte neue Kosten k gibt es mehrere Kombinationsmöglichkeiten von $i + j$, um diese zu erreichen, so z.B. die Kosten-Kombinationen $1+5$, $2+4$ und $3+3$ für die Kosten 6. Bezogen auf unsere Listen heißt das, dass wir für ein festes k jeweils zwei Listen $K[i]$ und $K[j]$ mit $i + j = k$ miteinander *kreuzen*. Kreuzen zweier Listen schreiben wir als K[i]$\oplus$ K[j] und meinen damit, dass alle Elemente $a \in K[i]$ mit allen Elementen $b \in K[j]$ paarweise via $a \oplus b$ kombiniert werden. In Abb. 2.3 ist schematisch gezeigt, wie neue Listen aus vorherigen Listen via Kreuzen hervorgehen. Abbildung 2.4 zeigt ein Beispiel, bei dem $K[2]$ vollständig berechnet wird.

2.2.1 Umgang mit verschiedenen Arten von Zahlen

Bei der Berechnung von neuen Listen $K[i]$ begegnet man nicht nur (kleinen) natürlichen Zahlen N, sondern auch mehreren neue Arte von Zahlen:

- Negative, (-20, ...)
- Zu große, ($\sim 10^{10}$)
- Konkatenierte, (44, 444, ...)
- Nicht-Ganzzahlige, (1/4, 1/6, ...)
- und bereits entdeckte Zahlen (4, 24, ...).

Der Umgang mit diesen verschiedenen Arten von Zahlen ist eine zentrale Komponente der Aufgabe. Er hat entscheidenden Einfluss auf Geschwindigkeit, Speicherverbrauch und Qualität des Algorithmus. Wir wollen deshalb jede Art einzeln etwas genauer diskutieren. Insbesondere

$$
\begin{aligned}
K[2] =& K[1] \oplus K[1] \\
=& \{4\oplus 4, \quad 4\oplus 24, \quad 24\oplus 24\} \\
=& \{4+4, \quad 4+24, \quad 24+24, \\
& \ 4*4, \quad 4*24, \quad 24*24, \\
& \ 4-4, \quad 4-24, \quad 24-24, \\
& \ 4-4, \quad 24-4, \quad 24-24, \\
& \ 4/4, \quad 4/24, \quad 24/24, \\
& \ 4/4, \quad 24/4, \quad 24/24, \\
& \ 4\text{\^{}}4, \quad 4\text{\^{}}24, \quad 24\text{\^{}}24, \\
& \ 4\text{\^{}}4, \quad 24\text{\^{}}4, \quad 24\text{\^{}}24\} \\
=& \{8, \quad 28, \quad 48, \\
& \ 16, \quad 96, \quad 576, \\
& \ 0, \quad -20, \quad 0, \\
& \ 0, \quad 20, \quad 0, \\
& \ 1, \quad 1/6, \quad 1, \\
& \ 1, \quad 6, \quad 1, \\
& \ 256, \quad \sim 2.8\cdot 10^{14}, \quad \sim 1.33\cdot 10^{33}, \\
& \ 256, \quad 331776, \quad \sim 1.33\cdot 10^{33}\}
\end{aligned}
$$

Fakultät: $8! = 40320$, $6! = 720$

Konkatenierte: 44

Aussortiert: $\{0, -20, \sim 2.8\cdot 10^{14}, \sim 1.33\cdot 10^{33}, n!\ (\forall\, n \geqslant 13)\}$

$$\Rightarrow K[2] = \{8,28,48,16,96,576,20,1,1/6,6,256,331776,40320,720,44\}$$

Abbildung 2.4: Berechnung der Liste $K[2]$.

wird es um die Frage gehen, wann mathematisch erlaubte Zahlen aus praktischen Gründen ignoriert werden können oder müssen, und wann es sich stattdessen lohnt, auch bei erhöhtem Aufwand mit ihnen zu rechnen. Abb. 2.4 zeigt ein Beispiel, bei dem einige neu berechnete Zahlen aussortiert werden und nicht in die finale Liste $K[2]$ übernommen werden, obwohl sie prinzipiell mit Kosten 2 erreichbar wären.

2.2.1.1 Negative Zahlen

Unsere Beobachtung zu negativen Zahlen ist: In Teilaufgabe a) bringen sie keinen zusätzlichen Nutzen. Negative Zahlen können dort ignoriert und aussortiert werden. Aussortieren heißt in unserem Kontext, dass, wenn sie das Ergebnis einer neuen Rechnung sind, sie nicht in ein Liste $K[i]$ gespeichert werden. Bei Teilaufgabe b) sehen wir eine kleine theoretische Möglichkeit, dass negative Zahlen einen Nutzen haben könnten, siehe Abschnitt weiter unten. Diese Chancen schätzen wir aber so gering ein, dass wir auch bei Teilaufgabe b) zum Einsparen von Laufzeit und Speicher negative Zahlen aussortieren. Das betrifft z.B. die Zahl -20 im Zuge der Berechnung von K[2]; sie wird ausgerechnet, aber nicht in K[2] gespeichert (siehe Abb. 2.4).

Der Nicht-Nutzen der negativen Zahlen (bei Teilaufgabe a) liegt in dem beschränkten Set an erlaubten Operationen $\{+,-,*,/\}$. Diese führen dazu dass die negativen Zahlen nur eine Spiegelung der positiven Zahlen darstellen, ohne dass sich neue Möglichkeiten für Rechnungen öffnen. Nimmt man die allgemeinste Rechnung $c = a \oplus b$ mit $\oplus \in \{+,-,*,/\}$ und nimmt an, dass a und/oder b negativ wären, dann kann man sie immer durch positive a,b ersetzen und entweder exakt das selbe Ergebnis reproduzieren oder das selbe Ergebnis mit umgekehrtem Vorzeichen, siehe Abb. 2.5. In ersteren Fällen waren die negativen Zahlen nicht nötig, in zweiteren Fällen kann ein negatives Ergebnis immer in ein positives umgekehrt werden, wodurch für folgende Rechnungen gesichert ist, dass zu jeder negativen Zahl $-c < 0$ zu den selben Kosten oder günstiger die positive gespiegelte Version c existiert.

$a,b > 0$

Rechnung	$\Rightarrow$	**Ersatz**	**Auswirkung**
$a+(\text{-}b)$		$a-b$	selbes Ergebnis
$(\text{-}a)+(\text{-}b)$		$a+b$	geflippt
$a-(\text{-}b)$		$a+b$	selbes Ergebnis
$(\text{-}a)-b$		$a+b$	umgekehrt
$(\text{-}a)-(\text{-}b)$		$b-a$	selbes Ergebnis
$a\cdot(\text{-}b)$		$a\cdot b$	umgekehrt
$a/(\text{-}b)$		a/b	umgekehrt
$(\text{-}a)\cdot(\text{-}b)$		$a\cdot b$	selbes Ergebnis
$(\text{-}a)/(\text{-}b)$		a/b	selbes Ergebnis
$(\text{-}a)\text{^}b$		$a\text{^}b$	umgekehrt
$a\text{^}(-b)$		$(1/a)\text{^}b$	**nur Kehrbruch**

Abbildung 2.5: Gedankenexperiment, warum negative Zahlen (bis auf bei der Potenzfunktion) überflüssig sind. Links: Hypothetische mögliche Rechnungen mit negativen Zahlen (-a) und (-b). Rechts: Jeweils eine Ersatzrechnung, die nur aus positiven Zahlen a und b besteht.

Bei der Potenz-Operation hingegen ist es vertrackter. Hier findet beim Austauschen eines negativen Exponenten $(-b)$ zu b keine Vorzeichen-Umkehrung statt, sondern man erhält den Kehrbruch $1/c$ des ursprünglichen Ergebnisses. Innerhalb der Operationen $*, /$ kann zwar beliebig $1/c$ durch c und andersrum kompensiert werden, aber bei den Operationen $+, -$ ist das nicht der Fall. Hier sind c und $1/c$ zwei sehr verschiedene Zahlen. In der Theorie wäre es also denkbar dass man einen negativen Exponenten anwendet und mit dem resultierenden Kehrbruch via Addition oder Subtraktion weiter rechnet, und so neue Zahlen günstiger erreichen kann. In unserer Implementation haben wir negative Zahlen jedoch aussortiert, dadurch konnten wir signifikant Laufzeit und Speicherplatz sparen. Sollte eine Einsendung aber zeigen können, dass mithilfe von negativen Zahlen günstigere Ergebnisse möglich sind, wäre so eine Untersuchung sehr zu begrüßen.

2.2.1.2 Die Null

Mit der Null kann nicht nützlich weitergerechnet werden, deshalb sortieren wir sie aus. Eine einzige Ausnahme wäre die Rechnung $(4-4)! = 0! = 1$ gewesen um günstig die Zahl 1 zu erreichen. Aber da die Geteilt-Operation erlaubt ist, geht das auch zu den selben Kosten mit $4/4 = 1$ (und analog für alle übrigen Ziffern), sodass kein weiterer Nutzen für die Null besteht.

2.2.1.3 Zu große Zahlen

Sehr schnell gerät man bei dieser Aufgabe an das Problem, dass im Prinzip beliebig große Zahlen generiert werden können, aber man in der Praxis nur begrenzten Speicherplatz zur Verfügung hat. Die Wahl einer oberen Schranke ist für sehr große Zahlen unvermeidbar. Eine rein technische Limitierung macht sich z.B. schon bei der Berechnung der allerersten Liste $K[2]$ (Abb. 2.4) bemerkbar, bei der Zahl 3311776. Nach Regeln der Aufgabenstellung wäre 3311776!, die Fakultät von 3311776, eine valide Zahl. Sie hat aber ca. 20 Millionen Stellen und einen Speicherverbrauch von 0,75 MB. Es ist klar dass solche großen Zahlen nicht mit einem praktisch funktionierendem Programm vereinbar sind. Zudem gibt die Aufgabenstellung implizit eine bestimmte Größenordnung der Zielzahlen von ca. 10^3 (2020,2030,2080 und 2980) vor. Man kann sich daran orientieren und abschätzen dass das Programm für Zielzahlen von 10^4 und auch noch 10^5 im allgemeinen funktionieren sollte, aber jenseits dessen ab ca. 10^6 aus praktischen Gesichtsgründen eine Grenze gezogen werden darf.

Ein bisschen Luft nach oben ist von Vorteil, denn eine kleine Zielzahl kann manchmal sehr günstig durch das Teilen von zwei größeren Zahlen erreicht werden. Um abzuschätzen wie viel Luft nach oben nötig bzw. nützlich ist, haben wir für die vier vorgegeben Zielzahlen und alle neun Startziffern Tests durchgeführt bei denen wir die obere Schranke variiert haben. Es stellte sich heraus dass das Variieren der oberen Schranke zwischen 10^9 und 10^6 keinen Unterschied in den erzielten Kosten machte. Erst als die Schranke auf 10^5 verringert wurde haben sich die gefundenen Kosten leicht erhöht, also die Lösungen verschlechtert. Daraus kann man abschätzen dass ca. drei Größenordnungen über der Zielzahl eine sinnvolle obere Schranke ist, im Fall der Aufgabenstellung also 10^6. Wir haben bei unserer Implementierung einen Integer 32-Bit-Datentyp genutzt, dieser konnte Werte bis zu $2^{31} - 1 \approx 2 \cdot 10^9$ darstellen. Das hat dann auch unsere obere Schranke bestimmt, die wir zu $2 \cdot 10^9$ gewählt haben. Während den Rechnungen haben wir die Zahlen kurzzeitig zum längeren Datentyp *Long* (64-Bit) konvertiert, dadurch

konnten Buffer-Overflows besser abgefangen werden die ansonsten beim Verrechnen von zwei 32-Bit Zahlen bei z.B. dem Benutzen der Potenz-Operation aufgetaucht wären.

Anmerkung: Der benötigte Speicherplatz hängt stark vom gewählten Algorithmus ab. Es kann andere Algorithmen geben, die deutlich weniger Zwischenspeicher benötigen und dadurch höhere obere Schranken erlauben. Das wäre z.B. bei einem heuristischen Ansatz der Fall, der sich aktiv und gezielt auf die Zielzahl zubewegt und dadurch mit sehr viel weniger zwischengespeicherten Werten auskommt. Im Gegensatz könnte so eine Heuristik ihrer Natur nach aber nicht mehr eine günstigst-mögliche Darstellung garantieren. Diese Abwägung (sofern sich über Heuristiken Gedanken gemacht wurden) muss jede Einsendung für sich selbst entscheiden. Unser Ansatz garantiert (im Rahmen der restlichen Einschränkungen) günstigst-mögliche Terme, ist dafür aber maximal speicherintensiv. Für die verlangten Zielzahlen war das aber kein Hindernis, dort blieb der Speicherverbrauch handhabbar.

2.2.1.4 Konkatenierte Zahlen

Laut Aufgabenstellung dürfen konkatenierte Zahlen wie 44, 444, 4444 per Hand generiert werden. Für uns heißt das, dass wir sie per Hand zu den jeweiligen Listen K[i] hinzufügen, wie es z.B. bei der Liste $K[2]$ für die Zahl 44 geschehen ist. Das ist eigentlich auch schon alles, sie sind ein kleiner Sonderfall, den man mit wenigen Zeilen Code behandeln kann. Worauf wir aber noch aufmerksam machen wollen ist folgendes Detail: Die konkatenierte Darstellung für eine Zahl ist nicht immer unbedingt auch die günstigste Darstellung. Man sieht das am Beispiel in Abbildung 2.6.

$$999999999 = (9 + 9/9)^9 - 9/9$$

Abbildung 2.6: Eine konkatenierte Zahl kann durch einen anderen Term günstiger dargestellt werden.

Leider (oder glücklicherweise, je nachdem wie man es empfindet) ist dieses Beispiel nach unseren Berechnungen aber auch der einzige Fall, bei dem für eine konkatenierte Zahl ein günstigerer Term existiert. Für alle anderen konkatenierten Zahlen mit bis zu neun Ziffern (wir haben bis zu einer Ziffernlänge von neun geprüft) ist über alle erlaubten Startziffern hinweg die konkatenierte Darstellung auch immer gleichzeitig die günstigste oder es es gibt es zumindest keine noch günstigere.

Wir haben hier also folgenden Fall: Die Idee "Konkatenierte Zahlen durch andere Terme günstiger darstellen"war von der Aufgabenstellung her komplett erlaubt, aber sie war hier zufälligerweise von keinem Nutzen. Wären die Rechenregeln etwas anders gewesen (z.B. zusätzliche Operationen erlaubt) dann hätten sich sehr wahrscheinlich solche günstigeren Terme für konkatenierte Zahlen finden lassen, und Teilnehmer die das erkannt hätten hätten einen Vorteil gehabt. Aber so wie die Rechenregeln dieser Aufgabe zufällig nun einmal gesetzt waren ließen sich keine günstigeren Terme generieren.

Wenn man dieses Detail übersehen hatte hat man somit Glück gehabt, es hatte keine Auswirkungen auf die Rechnungen. Falls man es trotzdem behandelt hat ist das natürlich zu begrüßen, insbesondere falls man auch das eine oben gezeigte Beispiel gefunden hat.

2.2.2 Nicht-ganzzahlige Zwischenergebnisse (Brüche)

Bisher haben wir nur ganze Zahlen ($\mathbb{Z}$) behandelt. Durch die Geteilt-Operation können wir aber eine weitere Klasse an Zahlen erreichen, die rationalen Zahlen $\mathbb{Q}$. Das erste Beispiel davon sehen wir bereits bei der Berechnung von $K[2]$ in Abbildung 2.4 gesehen, beim Bruch $4/24 = 1/6 \approx 0.166666666666667$. Bei Brüchen zeigt sich: Es lohnt sich, sie abzuspeichern und mit ihnen weiter zu rechnen. In Tabelle 2.2 sind drei Fälle zur Teilaufgaben a) und b) gezeigt, bei denen günstigere Terme gefunden werden konnten, sobald zusätzlich Brüche als Zwischenergebnisse erlaubt wurden. Falls man Brüche nicht erlaubt, wird das vermutlich oft durch Abfragen wie $a\%b == 0$ geschehen, bei denen dann $c = a/b$ nur dann ausrechnen wird wenn es wieder eine ganze Zahl ergibt.

Um noch einmal den Unterschied zu verdeutlichen: Ein Term „ohne Brüche“ kann durchaus die Geteilt-Operation enthalten. Aber alle inneren Klammerungen werden sich zu ganzen Zahlen auswerten. Bei Termen „mit Brüchen“ dagegen kann ein geklammerter Ausdruck sich zu einem Bruch auswerten, der nicht weiter kürzbar ist.

Der Nutzen von Brüchen liegt darin, dass sie sich im Laufe der Rechnung wieder rauskürzen können und damit zwischenzeitlich eine zusätzliche Freiheit erlauben, die Abkürzungen ermöglicht. Es stellt sich also nun die Frage, wie man das Rechnen mit Brüchen implementieren kann. Es bieten sich im Wesentlichen zwei Techniken an.

Man kann einen Bruch, z.B. $1/6$, entweder „ausgerechnet“ durch eine Gleitkommazahl c darstellen, $c = 0.166666666666667$, oder durch zwei natürliche Zahlen a und b, nämlich Zähler und Nenner des Bruchs, die z.B als Tupel $[a,b] = [1,6]$ gespeichert werden. Der zweite Ansatz ist derjenige, den wir gewählt haben. Denn Gleitkommazahlen haben das Problem von Rundungsfehlern, die beim Rechnen und Abspeichern wegen der endlichen Maschinengenauigkeit[1] auftauchen. Diese Ungenauigkeit sieht man z.B. bereits im obigen Ausdruck $0.166...667$ wo der Computer die letzte Ziffer notgedrungen zu einer 7 gerundet hat. Das Runden wird insbesondere ein Problem, wenn wiederholt gerechnet und gespeichert wird; dann können sich Fehler fortpflanzen und signifikant verstärken.

Solche Rundungsprobleme können fast komplett (bis auf bei der Potenzrechnung) umgangen werden wenn man die Repräsentation über zwei natürliche Zahlen a und b wählt. Die Rechnungen reduzieren sich dann (bis auf bei der Potenz-Operation) auf Multiplikation und Addition von natürlichen Zahlen, das kann exakt gerechnet werden. Der Preis, den man zahlt, sind mehr Rechnungen, z.B. drei Multiplikationen und eine Addition von zwei natürlichen Zahlen, um die Addition von zwei Brüchen durchzuführen, siehe Abb. 2.7. Dafür gewinnt man, dass jede einzelne Rechnung wieder exakt ist.

Zusätzlich sollten die resultierenden Brüche gekürzt werden, um zu verhindern dass man den mathematisch selben Bruch in zwei verschiedenen Darstellungen speichert. Wir benutzen den euklidischen Algorithmus und berechnen den größten gemeinsamen Teiler (ggT) eines Bruches $[a,b]$, und teilen dann jeweils a und b durch den ggT. Damit erhält man einen eindeutigen, maximal gekürzten Bruch.

Die Potenz-Funktion ist etwas schwieriger zu handhaben. Hier lässt sich eine gewisse Maschinenungenauigkeit nicht immer vermeiden. In Java erhält man z.B durch den Befehl *Math.pow(a,b)* die Ergebnisse $125^{1/3} = 4.999999999999999$ und $27^{2/3} = 8.999999999999998$ statt der korrekten Ergebnisse 5 und 9. Solche numerischen Fehler können erkannt werden daran, dass die

[1] `https://de.wikipedia.org/wiki/Maschinengenauigkeit`

	Mit Brüchen	**Ohne Brüche**
Teilaufg. a)	$2030 = \underbrace{\left(8(8+8) - \underbrace{\frac{8+\frac{8}{8}}{8}}_{\mathbf{9/8}}\right)}_{\mathbf{1015/8}}(8+8)$ **Kosten 9**	$2030 = 8(8+8)(8+8) - \left(8 + \frac{88-8}{8}\right)$ **Kosten 10**
Teilaufg. b)	$2020 = \underbrace{\left(\underbrace{\frac{7!}{7\cdot 7}}_{\mathbf{720/7}} - 7\right)}_{\mathbf{671/7}}(7+7+7)+7$ **Kosten 8**	$2020 = (7(7+77)-7) + \frac{7!+7!-7}{7}$ **Kosten 9**
Teilaufg. b)	$2980 = \underbrace{\left(\underbrace{\frac{7+7!+7!}{7\cdot 7}}_{\mathbf{1441/7}} + 7\right)}_{\mathbf{1490/7}}(7+7)$ **Kosten 8**	$2980 = \left(7! - \frac{7+7}{7}\right) - (7\cdot 7)(7\cdot 7-7)$ **Kosten 9**

Tabelle 2.2: Vergleich zwischen Zulassen (links) und Nicht-Zulassen (rechts) von Brüchen als Zwischenergebnisse. Es zeigt sich, dass Brüche zu günstigeren Termen führen können.

Implementierung:	Formel:
$[a,b] \pm [c,d] = [ad \pm bc, bd]$	$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$
$[a,b] \cdot [c,d] = [ac, bd]$	$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$
$[a,b] \div [c,d] = [ad, bc]$	$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$
$[a,b]\text{^}[c,d] = [a^{c/d}, b^{c/d}]$	$\left(\frac{a}{b}\right)^{c/d} = \frac{a^{c/d}}{b^{c/d}}$
$a,b,c,d \in \mathbb{N}$	

Abbildung 2.7: Implementierung von Bruchrechnung.

Resultate extrem nah an einer ganzen Zahl liegen, viel näher als es durch Zufall geschehen könnte. Mittels eines entsprechenden Toleranzbereiches (z.B. 10^{-13}) lassen sich diese minimalen Abweichungen erkennen und das Ergebnis dann auf eine korrekte ganze Zahl runden. Oft weichen die Ergebnisse nur um einen einzigen Schritt in der Maschinengenauigkeit ε vom tatsächlichen Ergebnis ab, das lässt sich in Java z.B. mit dem Befehl *Math.nextUp(a)* prüfen. Dieser gibt in Schritten der Maschinengenauigkeit die nächsthöhere Zahl aus. Angewandt auf die obigen Beispiele erhält man z.B. Math.nextUp(8.999999999999998)=9 und somit die Bestätigung, dass die Rechnung $27^{2/3}$ tatsächlich eine ganze Zahl ergibt und gespeichert werden kann.

Sollte das Resultat des Potenzierens außerhalb des Toleranzbereichs liegen, also das Ergebnis für $a^{(c/d)}$ oder $b^{(c/d)}$ irgendeine reelle, nicht-ganze Zahl sein, dann wird dieses Ergebnis bei uns aussortiert. Es besteht dann keine echte Chance mehr, von einer beliebigen reellen Zahl in wenigen Schritten wieder zurück zu den natürlichen Zahlen zu gelangen, wie es am Ende für die Zielzahl aber nötig wäre. Zudem würden reelle Zahlen wieder das Problem mit dem Runden einführen, was sich dann auch nicht mehr mit einer Darstellung wie $[a,b]$ mit $a,b \in \mathbb{N}$ retten ließe.

2.2.3 Hashmap: Duplikate verhindern

Beim Generieren neuer Zahlen stößt man oft auf bereits bekannte Zahlen, die man schon einmal generiert und gespeichert hat. Man interessiert sich aber nur für eine Darstellung mit den niedrigsten Kosten, die teureren Darstellungen möchte man auf jeden Fall vermeiden. Es ist deshalb sehr hilfreich Buch darüber zu führen, welche Zahlen (zu welchen Kosten) man bereits gefunden hat. In solch einem Buch kann man dann für jede neu generierte Zahl nachgeschlagen, ob sie bereits früher einmal gefunden wurde. Falls nein, kann sie nun eingetragen werden, falls ja, ist sie von keinem zusätzlichen Nutzen und kann ignoriert werden. In unserem Fall müssen solche Einträge in dem Buch nie aktualisiert werden, da die konstruierte Reihenfolge der generierten Kosten bereits sicherstellt, dass der erste Eintrag je Zahl ihre günstigst-möglichen Kosten enthält.

Eine Hashmap[2] (manchmal auch Hashtabelle genannt) bietet sich an. Sie ist eine Datenstruktur, die das Speichern und Suchen von Einträgen in amortisiert konstanter Zeit $\mathcal{O}(1)$ erlaubt. Insbesondere wegen der konstanten Zeit beim Suchen ist die Hashmap von Vorteil gegenüber z.B. den Listen $K[i]$, bei denen eine Suchoperation durchschnittlich lineare Zeit $\mathcal{O}(n)$ benötigt. In einer Hashmap können im Allgemeinen (key,data)-Paare gespeichert und gesucht werden. In z.B. Java gibt es dafür die Befehle *hashmap.put(key,data)* und *hashmap.get(key)*. Das Benutzen einer Hashmap ist fast genauso einfach wie das Benutzen eines üblichen Arrays, man muss nur die neue Syntax herausfinden.

Für unsere Zwecke wird die Hashmap folgendermaßen genutzt: Als key benutzen wir die generierten Zahlen selbst. Für jede Zahl ergibt sich dadurch in der Hashmap ein zugehöriger „Slot“, in dem Informationen zur Zahl gespeichert werden können. Als Information speichern wir die Kosten k der Zahl sowie die Zahlen a, b und die verwendete Operation aus der sie entstanden ist. Diese vier Werte speichern wir zusammen als *Historie* $[a,b,op,k]$. Ein Beispiel vom Speichern so einer Historie ist in Abb. 2.8 gezeigt.

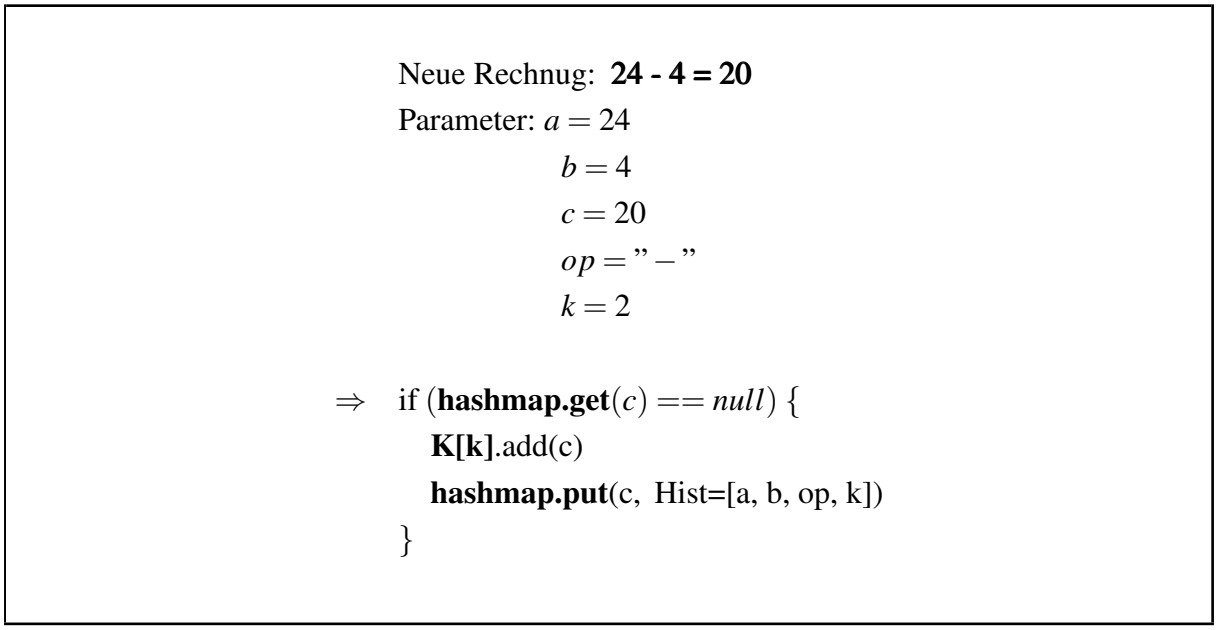


Abbildung 2.8: Speichern einer neuen Zahl in Hashmap und Liste $K[i]$.

Technische Anmerkung: Wir speichern alle Zahlen wie Brüche, also als Paar $[m,n]$ mit zwei 32-Bit-Integers m,n; für ganze Zahlen ist $n = 1$. Wir bilden dann einen key, indem wir m und n zu einer einzigen 64-Bit Zahl $(m << 32|n)$ konkatenieren, d.h. die erste Hälfte des key bestehen aus den Bits von m und die hintere Hälfte aus den Bits von n. Dieser so generierte 64-Bit key ist dann eindeutig für jede Zahlenkombination $[m,n]$. In Abb. 2.8 wurde zur besseren Lesbarkeit die Zahl 20 auch als 20 geschrieben, im Code wird sie als Tupel $[20,1]$ gehandhabt.

Eine Hashmap ist vergleichsweise speichereffizient, weil sie den Speicher dynamisch vergrößert, wenn neue Einträge hinzukommen. Das ist anders als z.B. bei einem zweidimensionalen Array A[a][b], das zwar von der Bedienung her den selben Nutzen bringen würde wie eine Hashmap, aber seinen gesamten Speicher bereits bei der Initialisierung allokiert. Das würde bei $a,b \in \{0..10^9\}$ und 64 Bit je Slot ein Speicherverbrauch von 10^6 TB erzeugen, oder 800 GB

[2] `https://de.wikipedia.org/wiki/Hashtabelle`

falls man die Nenner auf $b \leqslant 100$ beschränkt. Eine Hashmap wächst dagegen dynamisch, von einer Größe nahe Null bis in unserem Fall ca. 500 MB.

Eine Hashmap kann durch ein normales Array aber dann ersetzt werden, wenn man sich bei der oberen Schranke auf 10^6 beschränkt und nur ganze Zahlen statt Brüchen speichert, dann kann das resultierende Array eine Größe von unter 100 MB haben. Sollten sich Einsendungen also auf entsprechend kleinere Suchräume beschränkt haben, reicht auch ein normales Array statt einer Hashmap.

Fassen wir zusammen, wie die Hashmap verwendet wird: Zu Beginn ist sie leer. Jedes Mal, wenn eine neue Zahl mittels einer neuen Darstellung generiert wird, wird geprüft, ob für die Zahl schon ein Eintrag in der Hashmap existiert. Falls nicht, wird nun ein solcher Eintrag inklusive der gefundenen Darstellung (Historie) angelegt. Falls bereits ein Eintrag existiert, wird die neu gefundene Darstellung ignoriert und übersprungen. Es existiert dann schon eine günstigere bzw. genauso günstige Darstellung, die in der Hashmap gespeichert ist. Der Vorgang des Speicherns ist noch einmal in Abb. 2.8 gezeigt.

2.2.4 Term auslesen

Endgültiges Ziel der Aufgabe ist es, für die Zielzahl einen Term als String auszugeben. Das lässt sich basierend auf der bisherigen Vorarbeit leicht bewerkstelligen. Ist die Zielzahl c irgendwann gefunden, besitzen wir ihre Historie $[k,a,b,op]$. Der zugehörige Term T kann dann via $T =$ "(" $+ T_a + op + T_b +$ ")" zusammengestellt werden, wobei T_a und T_b die Terme für die Zahlen a und b sind. Sie werden rekursiv bestimmt, denn a und b besitzen wiederum ihre eigene Historie. Diese rekursive Struktur ist in Abb. 2.9 gezeigt. Der Basisfall ist dann erreicht, wenn ein neu aufgerufener Term nur noch aus der Startziffer selbst besteht.

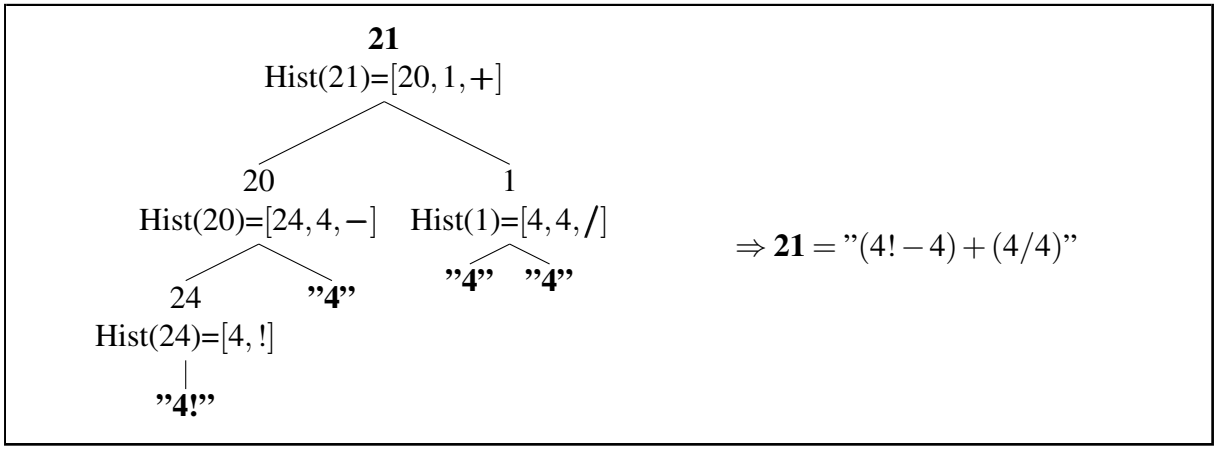


Abbildung 2.9: Auslesen eines Terms.

Für die Fakultätsfunktion wird der rekursive Aufruf entsprechend angepasst, da sie nur eine einzelne Zahl als Historie hat. Um minimalen Speicher zu verbrauchen, ist es am sparsamsten, für jede Zahl nur ihre Historie (anstatt des vollständigen Strings) zu speichern und ganz am Ende den gewünschten String rekursiv via den Historien zu rekonstruieren. Eine einfachere Version, die aber auch mehr Speicher benötigt, wäre, für jede Zahl sofort ihren gesamten Term als String zu speichern.

Der Algorithmus ist an dieser Stelle fertig erklärt. Es folgt noch ein kleiner optionaler Abschnitt, der eine mögliche Optimierung diskutiert. Danach werden die Ergebnisse für Teilaufgabe a) und b) gelistet.

2.2.5 Optimierung: Zielzahl rückwärts rechnen

Es gibt eine Optimierung für den bisher beschrieben Algorithmus. Diese besteht darin, von der Zielzahl aus „rückwärts“ zu expandieren, zusätzlich zum bisherigen Vorwärts-Expandieren von der Startziffer aus. Man generiert von zwei verschiedenen Fronten neue Zahlen und hat eine Lösung wenn diese Fronten aufeinandertreffen. Für eine bildliche Darstellung dazu siehe Abb. 2.11.

Start:	$\boldsymbol{N[0] = \{2019\}}$
Nachbarn über 1 Ziffer:	$2019 - 4 = 2015$
	$2019 + 4 = 2023$
	$2019 / 4 = 2019/4$
	$2019 \cdot 4 = 2076$
	$\Rightarrow \boldsymbol{N[1] = \{2015,\ 2023,\ 2019/4,\ 2076\}}$
Nachbarn über 2 Ziffern:	$2015 - 4 = 2011$
	$2019 - 16 = 2003$
	...

Abbildung 2.10: Optimierung: Es werden zusätzlich rückwärts Nachbarn der Zielzahl berechnet.

Der Ansatz ist, Nachbarn um die Zielzahl herum zu berechnen. Mit Nachbarn meinen wir Zahlen B, von denen aus man mit den geringsten weiteren Kosten zur Zielzahl gelangen kann. Diese weiteren Kosten nennen wir Vervollständigungskosten, kurz *VVS-Kosten*. Als Beispiel, angenommen die Zielzahl ist 2019 und die Startziffer ist 4, dann wären 2015 und 2023 Nachbarn mit VVS-Kosten 1, und 2011 und 2003 Nachbarn mit VVS-Kosten 2. In Abb. 2.10 sind diese ersten Berechnungen angedeutet. Die Arbeit bestehen nun darin diese Nachbarn zu finden. Wir speichern sie in Listen N[i], $i \geqslant 1$, aufgeteilt nach ihren VVS-Kosten. Alle Nachbarn mit VVS-Kosten i werden in Liste $N[i]$ gespeichert. Um den Algorithmus einheitlich zu halten, ist es sinnvoll, die Zielzahl in der Liste $N[0]$ zu speichern. Dann kann auf sie genauso wie auf die übrigen Nachbarn zugegriffen werden.

Die Nachbarn fungieren wie eine Zielscheibe um die Zielzahl herum: Mittels der bisherigen Expansion K[i] reicht es nun, auf eine einzigen bekannten Nachbarn zu stoßen, nennen wir ihn B, und schon hat man eine Lösung. Die Lösung setzt sich zusammen aus den beiden Teillösungen (Startziffer $\rightarrow$ B) und (B $\rightarrow$ Zielzahl), für die jeweils die Terme bekannt sind. Kombiniert ergibt sich die fertige Lösung (Startziffer $\rightarrow$ Zielzahl), die als Ergebnis für die Zielzahl ausgegeben werden kann.

Die Nachbarn B werden berechnet, indem man für feste A und C die Rechnung $A \oplus B = C$ nach B auflöst. Hierbei ist $C \in N[j]$ eine bereits bekannte Nachbarszahl mit VVS-Kosten j

und $A \in K[i]$ eine beliebige wie bisher generierte Zahl mit Kosten i. Die berechnete Zahl B wird bzgl. des Aussortierens genauso wie bisherige Zahlen behandelt; falls B zu groß, negativ oder nicht als Bruch darstellbar ist, wird sie aussortiert. Falls sie nicht aussortiert wird, sind ihre VVS-Kosten $j+i$, d.h. sie wird in der Liste N[i+j] gespeichert. In Tabelle 2.3 sind solche Berechnungen für $A = 4$ und $C = 2019$ gezeigt.

Szenario für B	**nach B aufgelöst**
$B+4=2019$	$B=2019-4$
$B-4=2019$	$B=2019+4$
$B\cdot 4=2019$	$B=2019/4$
$B/4=2019$	$B=2019\cdot 4$
$B^4=2019$	$B=2019^{(1/4)}$
$4^B=2019$	$B=ln(2019)\,/\,ln(4)$
$B!=2019$	Prüfen ob $2019 \in \{1!,...,13!\}$

Tabelle 2.3: Berechnung eines Nachbarn B.

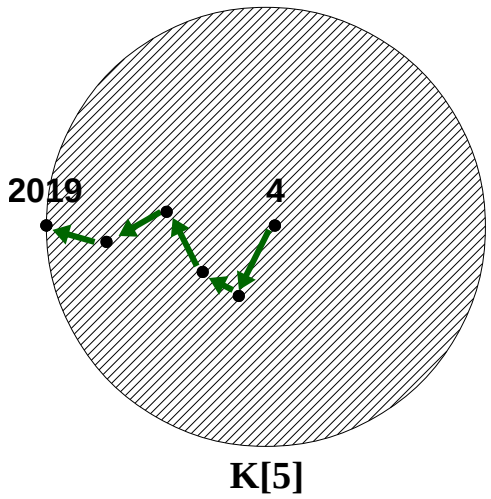
Der Vorteil der Optimierung liegt im Prinzip von Divide and Conquer[3]. Anstatt dass eine große Liste $K[10]$ berechnet wird, findet man schon eine Lösung, wenn man die beiden kleineren Listen $K[5]$ und $N[5]$ berechnet. Die Größe der Listen wächst exponentiell mit den Kosten, grob mit Faktor 4 je Kostenschritt: $|K[i]| \approx |N[i]| \approx i^4$. Die Listen $K[5]$ und $N[5]$ sind somit auch kombiniert sehr viel schneller berechnet als die eine große Liste $K[10]$.

Als Einschränkung muss erwähnt werden, dass effektiv noch ein paar Schritte in $K[i]$ weitergerechnet werden muss. Das liegt daran dass das erste Aufeinandertreffen der Listen $K[i]$ und $N[j]$ nicht unbedingt auch die günstigst-mögliche Lösung ergibt. Es kann sein als Erstes eine Lösung via Zusammentreffen von $K[7]$ und $N[3]$ gefunden wird (Kosten 10), aber später eine weitere Lösung via $K[8]$ und $N[1]$ gefunden wird, mit Kosten 9. Wenn also das erste Mal eine Lösung mit Kosten k entdeckt wird, müssen alle weiteren Kombinationen $i+j<k$ durchgerechnet werden, um sicherzustellen, dass keine günstigere Lösung übersehen wird. Hierbei ist immer $j \geqslant 1$, denn abgesehen von ausschließlich einzelnen konkatenierten Zahlen lässt sich jeder Term $C = A \oplus B$ in Teilterme $A \in K[i]$ und $B \in N[j]$ zerlegen. Es bleibt dann noch übrig $K[i]$ bis $i < k-1$ zu berechnen, also $i \leqslant k-2$. Gegenüber dem nicht-optimierten Verfahren bedeutet das, dass die letzten beiden Listen $K[k]$ und $K[k-1]$ nicht mehr berechnet werden müssen, um trotzdem mit Sicherheit ein günstigst-mögliches Ergebnis zu erhalten. Liegt also beispielsweise das günstigste Ergebnis bei Kosten 10, dann muss man in der optimierten Variante nur noch die beiden Listen $(K[8], B[8])$ statt das größere $K[10]$ berechnen.

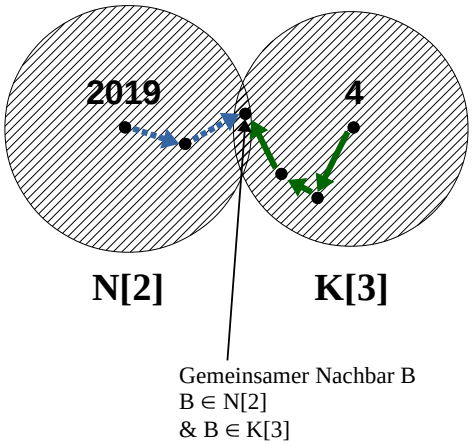
2.3 Trivia

Diese BwInf-Aufgabe ist eine relativ unerforschte Variante des „Four-Fours“-Problem [4]. Namensgebend für diese Art von Aufgaben, bei denen Ziffern mit Operationen verknüpft werden sollen, war eine frühe und populäre Variante, bei der genau vier Vierer verwendet werden sollten. Die zugelassenen Rechenregeln haben dabei je Autor variiert, für die Zielzahlen von 0 bis

[3] `https://de.wikipedia.org/wiki/Teile-und-herrsche-Verfahren`
[4] `https://en.wikipedia.org/wiki/Four_fours`



a) Bisheriger Algorithmus.



b) Optimierter Algorithmus: Zweiseitige Expansion.

Abbildung 2.11: Gegenüberstellung bisheriger Algorithmus (links) und optimierte Version (rechts). Die schraffierten Kreise symbolisieren die Menge an generierten Zahlen von einem Startpunkt ausgehend, bei der optimierten Version sind das insgesamt weniger.

10 reichen aber bereits die Grundrechenarten $(+, -, *, /)$ und Konkatenation, um eine Lösung zu finden.

```
0 = 4+4-4-4
1 = 4/4+4-4
2 = 4-(4+4)/4
3 = (4*4-4)/4
4 = 4+4*(4-4)
5 = (4*4+4)/4
6 = (4+4)/4+4
7 = 4+4-4/4
8 = (4/4)*4+4
9 = 4/4+4+4
10 = (44-4)/4
```

Abbildung 2.12: Eine historisch frühe Variante des ”Four-Fours”-Problems.

Seit Bekanntwerden des Rätsels haben sich insbesondere Privatpersonen dafür interessiert und in neuerer Zeit online Lösungen veröffentlicht [5] [6]. Dabei hat aber jeder Autor seine eigenen spezifischen Regeln aufgestellt, wodurch ein Vergleich der Lösungen schwierig ist. Es gibt eine in-

[5] https://dwheeler.com/fourfours/fourfours.pdf
[6] http://blog.nidi.guru/other/maths/2016/05/22/four-fours/

teressante Veröffentlichung von 2015, die erstaunlich nah an die Regeln dieser BwInf-Aufgabe herankommt[7]. Bei dieser wird auch versucht, die Anzahl an benötigten Ziffern zu minimieren, und die Menge der erlaubten Operationen entspricht beinahe exakt denen in dieser Aufgabe. In dem Paper werden Lösungsterme für alle Startziffern $1 \ldots 9$ und alle Zielzahlen $1 \ldots 1000$ ausgegeben. Damit liegen die dokumentierten Lösungen leider knapp unter dem für uns am interessantesten Bereich von 2019 bis 2980. Leider erwähnt der Autor mit keinem Wort seinen verwendeten Algorithmus, es werden nur Ergebnisse gelistet.

Nützliche Information zum algorithmischen Lösen des Four-Fours-Problems findet sich z.B. im englischen Wikipedia-Artikel[4], dort insbesondere zur Datenstruktur Hashmap, die auch wir für unsere Lösung benutzt haben. Vereinzelt findet man im Internet auch expliziten Code[8] [9], der eine Variante des FourFours-Problem löst. Diese Codes sind jedoch immer für bestimmte Regelwerke konzipiert und deshalb nicht für diese Aufgabe anwendbar. Die Berichte der Autoren über ihre Erfahrungen und Probleme beim Programmieren ihrer Aufgaben sind dagegen durchaus lesenswert, sie schildern z.B. explodierenden Speicherplatzverbrauch und Ungenauigkeiten im maschinellen Rechnen, also Probleme, denen auch wir begegnet sind.

Zum Lösen dieser Aufgabe gab es also kaum Literatur. Die meisten Informationen stammten von Privatpersonen, die sich in ihrer Freizeit mit dieser Aufgabe beschäftigt haben. Eine Ausnahme ist die Datenstruktur der Hashmap; sie war bei dieser Aufgabe sehr nützlich, und zu ihr findet man umfangreiche Informationen, denn sie ist eine der bekanntesten und verbreitetsten Datenstrukturen der Informatik. Sollte man sie trotzdem noch nicht gekannt haben, so konnte man über den Englischen Wikipedia-Artikel zu „Four-Fours“ darauf kommen – ein hilfreicher Tipp.

2.4 Ergebnisse

Teilaufgabe a) kann mit 339 Ziffern gelöst werden, Teilaufgabe b) mit 275 Ziffern. In Tabelle 2.4 ist eine Übersicht zu einigen weiteren Ergebnissen, die mit suboptimalen Einstellungen erreicht wurden. Verringert man z.B. die obere Schranke um zwei Größenordnungen auf 10^4 und speichert Brüche nicht mehr ab, so findet man für Teilaufgabe b) nur noch eine Lösung mit 285 Ziffern, d.h. man verbraucht dann 10 Ziffern mehr als eigentlich nötig wäre. Man sieht, dass selbst für starke Einschränkungen die Kosten sich relativ gesehen nur um 3-4 % ändern, d.h. der Unterschied zwischen verschiedenen Algorithmen wird sich hauptsächlich im Bereich von einstelligen Kosten-Änderungen bemerkbar machen.

Die gesamte Laufzeit für das Lösen der Teilaufgaben a) und b) liegt in unserem Fall bei a) 8,6 Sekunden und b) 3,4 Sekunden. Die Berechnungszeit für einzelne Zahlen variiert dabei jedoch stark, in einigen Fällen beträgt sie nur 0,002s, durchschnittlich ca. 0,2s und in manchen Fällen bis zu 1s. Eine Zeitmessung über eine gesamte Teilaufgabe gibt deshalb einen robusteren und aussagekräftigeren Wert.

```
Teilaufgabe a)
```

[7] https://arxiv.org/pdf/1502.03501.pdf
[8] http://writeasync.net/?p=5423
[9] http://blog.nidi.guru/other/maths/2016/05/22/four-fours/

Parameter	Gefundene Lösung Teilaufg. a)	Gefundene Lösung Teilaufg. b)
Obere Schranke 10^6, Brüche erlaubt	**339** Ziffern	**275** Ziffern
Obere Schranke 10^6, Brüche nicht erlaubt	340 (+1)	277 (+2)
Obere Schranke 10^4, Brüche erlaubt	340 (+1)	283 (+8)
Obere Schranke 10^4, Brüche nicht erlaubt	342 (+3)	285 (+10)

Tabelle 2.4: Übersicht zu den gefundenen Lösungen für Teilaufgaben a) und b). Es sind jeweils die Summe der verwendeten Ziffern angegeben. Die erste Zeile zeigt unsere beste gefundene Lösung. Die unteren drei Zeilen zeigen suboptimale Einstellungen (z.B. Schranke niedriger, keine Brüche erlaubt) und welche Auswirkungen diese haben. Sie führen zu erhöhten Kosten (+1 bis +10), d.h. bei den suboptimalen Einstellungen werden nicht mehr alle günstigsten Terme gefunden.

```
Zielzahl 2020
Ziff. 1: Kosten 10    2020 = (((1-((1+1)/11))+1)*1111)
Ziff. 2: Kosten 8     2020 = ((((2+(2*22))*22)-2)*2)
Ziff. 3: Kosten 9     2020 = ((((33/3)/3)+333)*(3+3))
Ziff. 4: Kosten 8     2020 = ((((4*(4*(4*4)))-4)*(4+4))+4)
Ziff. 5: Kosten 8     2020 = (((((55+(5*5))*5)+5)*5)-5)
Ziff. 6: Kosten 10    2020 = ((((6+((6+6)/6))/6)+66)*((6*6)-6))
Ziff. 7: Kosten 9     2020 = ((((7*((7*7)-7))-7)*7)+(77/7))
Ziff. 8: Kosten 9     2020 = (((8*((8*(8*8))-8))+8)/((8+8)/8))
Ziff. 9: Kosten 9     2020 = ((99+((9+9)/9))*(9+(99/9)))

Zielzahl 2030
Ziff. 1: Kosten 12    2030 = ((((1+1)*1111)+11)*(1-(1/11)))
Ziff. 2: Kosten 9     2030 = (((((2+(2*22))*22)+2)*2)+2)
Ziff. 3: Kosten 9     2030 = (((3+3)*333)+(33-(3/3)))
Ziff. 4: Kosten 10    2030 = (((4+(4+444))*(4+(4/(4+4))))-4)
Ziff. 5: Kosten 8     2030 = (((((55+(5*5))*5)+5)*5)+5)
Ziff. 6: Kosten 10    2030 = (((6*6)-((6+(6+(6/6)))/6))*(66-6))
Ziff. 7: Kosten 8     2030 = ((((7*((7*7)-7))+7)*7)-77)
Ziff. 8: Kosten 9     2030 = (((8*(8+8))-((8+(8/8))/8))*(8+8))
Ziff. 9: Kosten 10    2030 = (((9*9)-(99/9))*(9+(9+(99/9))))

Zielzahl 2080
Ziff. 1: Kosten 12    2080 = ((((111-11)-1)*((11+11)-1))+1)
Ziff. 2: Kosten 9     2080 = ((2*(2+(2+22)))*(2*(22-2)))
Ziff. 3: Kosten 9     2080 = (((3*(3+(3*(3+3))))*33)+(3/3))
Ziff. 4: Kosten 7     2080 = ((4+(4+44))*(44-4))
Ziff. 5: Kosten 8     2080 = ((5+(5*(5*5)))*(5+(55/5)))
Ziff. 6: Kosten 10    2080 = (((6*6)-((6+((6+6)/6))/6))*(66-6))
Ziff. 7: Kosten 9     2080 = ((((((7+7)*(7+7))-7)*77)+7)/7)
Ziff. 8: Kosten 8     2080 = (((8*8)+(8/8))*(8+(8+(8+8))))
Ziff. 9: Kosten 9     2080 = (((9+((9+99)/9))*99)+(9/9))
```

```
Zielzahl 2980
Ziff. 1: Kosten 13    2980 = ((((1+((1+1)*(1+(1+11))))*11)+1)*(11-1))
Ziff. 2: Kosten 11    2980 = ((((22+(2*(22-2)))*(2+22))+2)*2)
Ziff. 3: Kosten 9     2980 = ((((3*(333-3))+3)*3)+(3/3))
Ziff. 4: Kosten 9     2980 = (((((4*(4*44))-4)+44)*4)+4)
Ziff. 5: Kosten 8     2980 = ((55*55)-((55-5)-5))
Ziff. 6: Kosten 11    2980 = (((((6+6)*(6+(6*6)))-6)*6)-(6+((6+6)/6)))
Ziff. 7: Kosten 11    2980 = (((7*7)-(7-(((77/7)-7)/7)))*(77-7))
Ziff. 8: Kosten 11    2980 = (((((8*((8*(8+88))-8))+8)/(8+8))-8)*8)
Ziff. 9: Kosten 10    2980 = (((99*(9+(9+9)))+9)*((9+(9/9))/9))

Kosten Summe = 339
Laufzeit gesamt: 8.6 sec


----
Teilaufgabe b)

Zielzahl 2020
Ziff. 1: Kosten 10    2020 = (((1-((1+1)/11))+1)*1111)
Ziff. 2: Kosten 8     2020 = ((2^(22/2))-(2+(2+((2+2)!))))
Ziff. 3: Kosten 6     2020 = ((3*((3!)!))-((((3!)!)+(((3!)!)/(3!)))/(3!)))
Ziff. 4: Kosten 5     2020 = ((((4+4)!)/((4!)-4))+4)
Ziff. 5: Kosten 7     2020 = (((5!)+(5^5))-(5*(5+((5!)+(5!)))))
Ziff. 6: Kosten 7     2020 = (((6!)+((6!)+(6!)))-(((6!)+((6!)/6))/6))
Ziff. 7: Kosten 8     2020 = ((((((7!)/7)/7)-7)*(7+(7+7)))+7)
Ziff. 8: Kosten 9     2020 = (((8*(8+88))-8)+((((8!)+(8!))/8)/8))
Ziff. 9: Kosten 9     2020 = ((((9+((9!)/(99-9)))*9)-9)/(9+9))

Zielzahl 2030
Ziff. 1: Kosten 10    2030 = ((((1+1)^11)-1)-(11+((1+(1+1))!)))
Ziff. 2: Kosten 8     2030 = ((2^(22/2))-((22-2)-2))
Ziff. 3: Kosten 6     2030 = ((3*(((3!)!)-3))-(((3!)+((3!)!))/(3!)))
Ziff. 4: Kosten 7     2030 = ((4-(4/(4+4)))*(4+((4!)*(4!))))
Ziff. 5: Kosten 7     2030 = ((((555-5)-(5!))*5)-(5!))
Ziff. 6: Kosten 7     2030 = (((((6!)+(6^6))/6)/6)+((6!)-6))
Ziff. 7: Kosten 7     2030 = (((7+((7!)/(7+7)))-77)*7)
Ziff. 8: Kosten 8     2030 = ((88*(8+8))+((((8!)/8)/8)-8))
Ziff. 9: Kosten 8     2030 = ((((9+9)/9)^(99/9))-(9+9))

Zielzahl 2080
Ziff. 1: Kosten 10    2080 = ((11*(1+(1+1)))+(((1+1)^11)-1))
Ziff. 2: Kosten 8     2080 = (((2*((2+2)!))^2)-(2+222))
Ziff. 3: Kosten 5     2080 = ((3-(3/(3^3)))*((3!)!))
Ziff. 4: Kosten 5     2080 = ((4+(4^4))*(4+4))
Ziff. 5: Kosten 7     2080 = ((((5+555)-(5!))*5)-(5!))
Ziff. 6: Kosten 8     2080 = ((((6!)*((6!)+((6!)/6)))/(6*(6*6)))-(6!))
Ziff. 7: Kosten 9     2080 = (((((7+(7+7))/7)/7)*((7!)-7))-77)
Ziff. 8: Kosten 8     2080 = ((8*888)-((((8!)/8)-8)-8))
Ziff. 9: Kosten 8     2080 = ((9*(9*(9*9)))-((9+((9!)/9))/9))
```

```
Zielzahl 2980
Ziff. 1: Kosten 11    2980 = ((((1+((1+(1+1))!))!)-11)-(1+((1+1)^11)))
Ziff. 2: Kosten 8     2980 = (((2*((2+2)!))^2)+((2+((2+2)!))^2))
Ziff. 3: Kosten 7     2980 = ((3*(((3!)!)+33))+(((3!)!)+(3/3)))
Ziff. 4: Kosten 5     2980 = ((((4!)+(((4!)/4)!))*4)+4)
Ziff. 5: Kosten 5     2980 = ((((5*(5!))-5)*5)+5)
Ziff. 6: Kosten 8     2980 = ((((6*(6!))-(6!))-(6!))+(((6!)-((6!)/6))/6))
Ziff. 7: Kosten 8     2980 = (((((7+((7!)+(7!)))/7)/7)+7)*(7+7))
Ziff. 8: Kosten 9     2980 = (((8+(((8^8)/8)/8))+88)/88)
Ziff. 9: Kosten 9     2980 = (((99+((9!)/(9+9)))/9)+(9*(9*9)))

Summe Kosten = 275
Laufzeit gesamt: 3.4 sec


---
Auswahl der drei Terme die durch Brueche verbessert wurden.
a) 8 --> Kosten=9 2030 = (((8*(8+8))-((8+(8/8))/8))*(8+8))
b) 7 --> Kosten=8 2020 = ((((((7!)/7)/7)-7)*(7+(7+7)))+7)
b) 7 --> Kosten=8 2980 = (((((7+((7!)+(7!)))/7)/7)+7)*(7+7))
```

```
Beispiel 123456789

ohne Potenz & Fakultaet:
Ziff. 1: Kosten 15 123456789 = ((111111111/(1-(1/(11-1))))-1)
Ziff. 2: Kosten 17 123456789 = ((222222222/(2-(2/((22-2)/2))))-(2/2))
Ziff. 3: Kosten 16 123456789 = (((333333333*((3+(3/(3*3)))/3))-3)/3)
Ziff. 4: Kosten 17 123456789 = ((444444444/(4-(4/((44-4)/4))))-(4/4))
Ziff. 5: Kosten 15 123456789 = ((555555555/(5-(5/(5+5))))-(5/5))
Ziff. 6: Kosten 17 123456789 = ((666666666/(6-(6/((66-6)/6))))-(6/6))
Ziff. 7: Kosten 17 123456789 = ((777777777/(7-(7/((77-7)/7))))-(7/7))
Ziff. 8: Kosten 17 123456789 = ((888888888/(8-(8/((88-8)/8))))-(8/8))
Ziff. 9: Kosten 14 123456789 = ((((9*(9*(9+(9*(9+9)))))-9)*(9+(9*(999-9))))-9)
-
Summe Kosten: 145      (und im Fall wenn keine Brueche benutzt: 153)
Laufzeit gesamt: 50-60 sec


mit Potenz & Fakultaet:
Ziff. 1: Kosten 15 123456789 = ((111111111/(1-(1/(11-1))))-1)
Ziff. 2: Kosten 16 123456789 = ((2+((22*(2+222))/2))+(2+((22222/2)^2)))
Ziff. 3: Kosten 11 123456789 =
    (((3+((3!)*((3!)+(3*((3!)^(3!))))))*((3^3)+(((3!)!)/(3!))))-((3!)!))
Ziff. 4: Kosten 14 123456789 =
    (((((((4^4)*((4!)+(4!)))-(4!))/4)-4)*(((4+4)!)-(4/4)))+(44/4))
Ziff. 5: Kosten 14 123456789 =
    (((((5+5)!)-5)*(5+(5+((5!)/5))))+((((5!)*(((5!)+(5^5))-5))-5)/5))
Ziff. 6: Kosten 13 123456789 =
    ((((((6!)+(((6!)*((6!)+(6!)))-((6+6)/6)))-66)*((6!)-6))+6)/6)
Ziff. 7: Kosten 14 123456789 = ((((7*(7+(7+((7-(7/7))^7))))-7)*((77-7)-7))-((7!)/7))
Ziff. 8: Kosten 14 123456789 = (((((8!)-((88*(8+(((8!)/8)/8)))-(8!)))*((8!)-(8/8)))+88)/8)
Ziff. 9: Kosten 12 123456789 = (((((((9+(9/9))^9)-(9/9))/9)*(9+(9/9)))-9)/9)
```

```
-
Summe Kosten: 123      (und im Fall wenn keine Brueche benutzt: 124)
Laufzeit gesamt: 70-80 sec
```

2.5 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Es sind unterschiedliche Ansätze denkbar, an den Aufbau der Terme heranzugehen. Es ist alles in Ordnung, was dem Problem einigermaßen gerecht wird.

(2) *Ziffernanzahl gezielt minimiert*
- Ein systematischer, sukzessiver Aufbau von Termen, wie in der Beispiellösung, ist naheliegend und garantiert, dass die gefundenen Terme möglichst wenige Ziffern enthalten. Dies ist *die* zentrale Bedingung, in der Aufgabenstellung steht dazu „muss“. Heuristische Suchen können nur in Ausnahmefällen akzeptiert werden, falls sie klare Vorteile in Laufzeit und Größe der handhabbaren Zahlen erlauben und ggf. auch zu neuen interessanten Beobachtungen führen.
- Zwischenergebnisse müssen geeignet verwaltet bzw. gespeichert werden, so dass Duplikate mit mehr Termen vermieden werden.

(3) *Suchraum sinnvoll eingeschränkt*
- Auch bei dieser Aufgabe kann der Suchraum schnell sehr groß werden, insbesondere bei Teil b. Es ist also nötig und sinnvoll, den Suchraum einzuschränken. Das kann insbesondere durch „Aussortieren“ von Zwischenergebnissen passieren.
- *Negative Zahlen* liefern keinen Mehrwert und sollten aussortiert werden.
- *Zu große Zahlen* führen voraussichtlich nicht weiter. Eine obere Schranke für Zwischenergebnisse ist deshalb akzeptabel. Sie sollte aber nicht zu niedrig und mit guter Begründung gewählt worden sein.

(4) *Laufzeit des Verfahrens in Ordnung*
- Für alle vorgegebenen Werte können in kurzer Zeit Terme berechnet werden. Die Laufzeit sollte durch die Wahl einer geeigneten Datenstruktur zur Speicherung der Zwischenergebnisse befördert werden.
- Optimierungen der Termsuche können mit Pluspunkten belohnt werden, falls sie keinen negativen Einfluss auf die Ergebnisse haben können (Heuristiken sind hier also nicht gefragt).

(5) *Speicherbedarf in Ordnung*: Das Verfahren sollte nicht durch unnötig hohen Speicherbedarf ausgebremst werden.

(6) *Verfahren mit guten Ergebnissen*
- Die Terme müssen korrekt berechnet sein, also ausgerechnet die vorgegebene Zahl als Ergebnis haben.
- Bezüglich der Anzahl der in den gefundenen Termen enthaltenen Ziffern ist das Verfahren in der Regel genau so gut wie die Beispiellösung. Ausnahme sind (ggf.) die drei Fälle, in denen die Musterlösung durch das Rechnen mit Brüchen besonders kurze Terme findet. Für bessere Ergebnisse kann es Pluspunkte geben.

(7) *Brüche als Zwischenergebnisse*: Wenn Brüche als Zwischenergebnisse zugelassen werden, lassen sich in einigen Fällen kürzere Terme finden. Wenn Brüche (bzw. durch Division entstandene nicht-ganzzahlige Zwischenergebnisse) aussortiert werden, sollte das begründet werden; die Behauptung, dass Brüche nicht weiterhelfen, ist jedenfalls falsch. Die (korrekte) Behandlung von Brüchen bedeutet Zusatzaufwand und wird mit Bonuspunkten belohnt. Dabei sollten Brüche als Paare ganzzahliger Zähler und Nenner behandelt werden. Rechnen mit Gleitkommazahlen ist aber auch akzeptabel, wenn klar ist, dass dies in geeigneter Weise geschieht und die Ergebnisse nicht verfälscht.

2. Theoretische Analyse

(1) *Verfahren / Qualität ingesamt gut begründet*
- Es sollte ausdrücklich erklärt sein, wie die Bedingung von möglichst wenigen benötigten Ziffern umgesetzt wurde. Optimalität der Ergebnisse sollte nur behauptet werden, wenn die Suche ausschließlich optimalitätserhaltend eingeschränkt wird.
- Durch obere Schranken für Zwischenergebnisse, aber auch durch heuristische Verfahren kann verhindert werden, dass kürzestmögliche Terme gefunden werden. Dies sollte erkannt worden sein.

(2) *Gute Überlegungen zu Laufzeit und Speicher*: Präzise Angaben zur Laufzeit sind nur schwer möglich. Dennoch sollte sich die Einsendung zur Laufzeit und gerade bei dieser Aufgabe auch zum Speicherverbrauch Gedanken machen.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Zu allen vorgegebenen Jahreszahlen (2020, 2030, 2080 und 2980) sollten für Teil a (ohne Potenz und Fakultät) jeweils für alle Ziffern (1 bis 9) Terme angegeben werden. Für Teil b ist akzeptabel, wenn nur diejenigen Ziffern dokumentiert sind, für die sich kürzere Terme ergeben.

(5) *Ergebnisse nachvollziehbar dargestellt*: Man kann sich darüber streiten, ob die Ergebnisse nach Teilaufgaben (wie in der Beispiellösung) oder nach Jahreszahlen (das ist eigentlich noch schöner) gegliedert ausgegeben werden. Auf jeden Fall sollte aber zu jedem Term die Anzahl der darin enthaltenen Ziffern angegeben sein, damit man nicht mühsam nachzählen muss.

Aufgabe 3: Abbiegen?

3.1 Lösungsidee

Für die Aufgabe war ein Stadtplan mit einem designierten Start- und Endpunkt gegeben. Zwischen diesen sollten nun Wege bestimmt werden, deren Länge einen bestimmten Faktor von dem kürzesten Pfad nicht überschreitet. Unter diesen sollte dann ein Weg gefunden werden, der die Anzahl der Abbiegungen minimiert.

Wenn man sich die Aufgabenstellung anschaut, zerfällt sie logisch in zwei Teile.

1. Zwischen welchen Straßen findet eine Abbiegung statt?
2. Wie findet man, gegeben den Faktor und zwischen welchen Straßen eine Abbiegung stattfindet, einen möglichst abbiegungsfreien Weg?

Diesen Teilaufgaben werden wir uns jetzt nacheinander widmen.

3.1.1 Abbiegung?

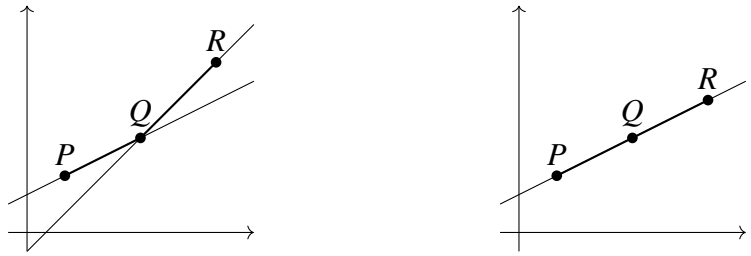
Nun wollen wir herausfinden, ob zwischen zwei gegebenen Straßen, die sich einen Mittelpunkt teilen, eine Abbiegung nötig ist. In der Aufgabenstellung werden nur gerade Straßen betrachtet. Diese lassen sich als Abschnitte linearer Gleichungen definieren, mit denen wir dann arbeiten können. Um senkrecht verlaufende Straßen nicht gesondert betrachten zu müssen, werden wir lineare Funktionen im Vektorraum über $\mathbb{R}^2$ betrachten.

Eine Straße wird durch zwei Punkte definiert. Betrachten wir nun eine Straße, welche durch die Punkte P und Q verläuft. Wenn O den Koordinatenursprung bezeichnet, dann kann man diese Straße als die lineare Funktion $s : [0,1] \mapsto \mathbb{R}^2$ mit $s(\tau) = \overrightarrow{OP} + \tau \cdot \overrightarrow{PQ}$ beschreiben. Da für das Argument τ nur Werte zwischen 0 und 1 erlaubt sind, ist der Wertebereich der Funktion genau die Punkte auf der Strecke $\overline{PQ}$, wie man in Abbildung 3.1 sieht.

(a) Straße

(b) Vektordarstellung

Abbildung 3.1: Visualisierung einer Straße als lineare Gleichung



(a) Abbiegung nötig (b) Abbiegung nicht nötig

Abbildung 3.2: Beispiel lineare Funktionen von Straßen

Betrachten wir nun den Übergang von der Straße A auf die Straße B. Diese Straßen verlaufen durch die Punkte P und Q bzw. Q und R. Ein solches Beispiel ist in Abbildung 3.2 visualisiert. Wir sehen dabei, dass eine Abbiegung notwendig ist, wenn R nicht auf der von P und Q beschriebenen Geraden liegt. Dies ist nicht der Fall, wenn R und Q von P aus in der gleichen Richtung liegen. Ferner muss R von P aus hinter Q liegen. Gibt es also ein $\tau \in \mathbb{R}_{>1}$ mit $\overrightarrow{OR} = \overrightarrow{OP} + \tau \cdot \overrightarrow{PQ}$, dann muss man nicht abbiegen.

Abbiegungsüberprüfung über eine Normalform Bei dieser Bedingung nehmen die beiden Straßen jeweils eine andere Rolle an. Zum einen gibt es die Straße, durch die die lineare Funktion bestimmt wird. Für die andere Straße muss dann geprüft werden, ob ihr neuer Punkt auf der linearen Funktion liegt. Um später eine Optimierung durchführen zu können, ist eine Formulierung besser, bei der die Straßen gleich behandelt werden. Betrachte dafür folgende Umformungen.

$$\overrightarrow{OR} = \overrightarrow{OP} + \tau \cdot \overrightarrow{PQ} \tag{3.1}$$

$$\iff \overrightarrow{OQ} + \overrightarrow{QR} = \overrightarrow{OQ} + \overrightarrow{QP} + \tau \cdot \overrightarrow{PQ} \tag{3.2}$$

$$\iff \overrightarrow{QR} = \overrightarrow{QP} + \tau \cdot \overrightarrow{PQ} \tag{3.3}$$

$$\iff \overrightarrow{QR} = (\tau - 1) \cdot \overrightarrow{PQ} \tag{3.4}$$

Da $\tau > 1$ ist muss in dieser Formulierung der Abbiegebedingung nur noch geprüft werden, ob die Richtungsvektor der beiden Straßen positive Vielfache voneinander sind.

Da alle Positionen in der Eingabe ganzzahlig sind, bietet sich eine weitere Vereinfachung an, die uns erlaubt die Richtungsvektoren in eine Normalform zu bringen und dann nur noch auf Gleichheit vergleichen zu müssen.

Zuerst gilt es dafür zu beobachten, dass damit auch alle Komponenten der Richtungsvektoren ganzzahlig sind. Seien $\vec{a}, \vec{b}$ nun ganzzahlige Vektoren mit einem $\lambda \in \mathbb{R}^+$ so, dass $\vec{a} = \lambda \cdot \vec{b}$ ist. Sei nun $\vec{c}$ ein ganzzahliger Vektor mit $\vec{c} = g_a \vec{a}$ und $g_a \in \mathbb{N}$ so, dass die Komponenten von $\vec{c}$ teilerfremd sind, also $\gcd(c_x, c_y) = 1$ gilt. Die Normalform, in die wir die Vektoren bringen wollen, ist für $\vec{a}$ nun $\vec{c}$. Diesen Vektor haben wir aus $\vec{a}$ erhalten, indem wir beide Komponenten durch ihren größten gemeinsamen Teiler – hier ist dieser g_a – geteilt haben.

Nun werden wir zeigen, dass $\vec{b}$ ein ganzzahliges Vielfaches von $\vec{c}$ ist. Dies garantiert uns, dass wir diese Tansformation auch auf $\vec{b}$ anwenden können und dabei die gleiche Normalform erhalten.

Nun muss es also ein $\lambda' \in \mathbb{R}^+$ so geben, dass $\vec{c} = \lambda' \cdot \vec{b}$ gilt. Ferner muss $\lambda' \in \mathbb{Q}^+$ gelten, da sonst $b_x\lambda' \notin \mathbb{Q}$ und damit ungleich c_x wäre. Sei also $\lambda' = \frac{m}{n}$ ein unkürzbarer Bruch. Nun ist $c_x = \frac{m}{n}b_x$ und damit $\frac{n}{m}c_x = b_x$. Demnach muss $m \mid nc_x$[1] gelten. Analog gilt $m \mid nc_y$. Da m und n teilerfremd sind, muss $m \mid c_x$ und $m \mid c_y$ gelten. Die Zahl m ist also ein Teiler von c_x und c_y. Es gilt $\gcd(c_x, c_y) = 1$, demnach ist also $|m| \leq 1$ und es gilt entweder $|m| = 1$ was zu $|n|\vec{c} = \vec{b}$ führt, oder $m = 0$, was zu $\vec{c} = \binom{0}{0} = \frac{m}{n}\vec{b}$ führt.

Ist $\vec{a}$ also nicht der Nullvektor, so haben wir gezeigt, dass die Überführung in die Normalform von $\vec{a}$ und $\vec{b}$ zu dem gleichen Ergebnis führt. Damit kann man diese nutzen, um auf Abbiegungen zu prüfen. Nun können wir zu jeder Kante den Richtungsvektor vorberechnen, die einzelnen Komponenten durch den größten gemeinsamen Teiler[2] teilen und diesen Wert dann zu dieser Kante abspeichern.

Dabei muss beachtet werden, dass die Orientierung, in der die Richtungsvektoren berechnet werden, einheitlich gewählt wird.

3.1.2 Abbiegungsarmen Weg finden

Die Aufgabe besteht darin einen möglichst abbiegungsarmen Weg finden, der zwischen zwei Knoten verläuft und dessen Länge dabei einen bestimmten Faktor der Länge des kürzesten Weges nicht überschreitet.

Zur Lösung des Problems bietet es sich an, zuerst die Länge des kürzesten Pfades[3] zwischen den beiden Punkten zu bestimmen. Dann kann man sich die Maximallänge des Pfades ausrechnen, den man Bilal ausgeben darf. Sollte man nun, gegeben eine Längenbegrenzung, den Pfad mit den wenigsten Abbiegungen finden können, so kann man auch das Ausgangsproblem lösen.

Aber auch dieses Problem lässt sich noch nicht gut lösen. Betrachten wir das Problem doch einmal anders herum. Sollten wir eine Anzahl an Abbiegungen gegeben haben, können wir dann die kürzeste Strecke finden, die höchstens so viele Abbiegungen verwendet? Es stellt sich heraus: Dieses Problem lässt sich viel besser lösen, da die Beschränkung diskretisiert wurde. Je nachdem wie die Berechnung erfolgt, kann man dann entweder eine lineare oder binäre Suche nach der mindestens benötigten Abbiegungszahl durchführen.

Um das Problem nun zu lösen, werden wir ein dynamisches Programm[4] angeben. Aber zuerst müssen wir formalisieren, über was wir eigentlich sprechen.

Wir werden die Straßenkarte als gewichteter, ungerichteter Graph $G = (V, E, w)$ formalisieren. Die Gewichtsfunktion $w : E \mapsto \mathbb{R}_0^+$ gibt dabei die euklidische Länge einer Kante an. Wir werden w auch für Pfade verwenden. Dann ist w als die Summe aller Kantengewichte definiert.

[1] $p \mid q$ bedeutet hierbei, dass p die Zahl q teilt. Formal also $\exists k \in \mathbb{Z} : kp = q$.

[2] Diesen kann man mit dem euklidischen Algorithmus bestimmen.

[3] Dafür kann bspw. Dijkstras Algorithmus genutzt werden. Wir werden später sehen, dass wir dies doch nicht machen müssen, aber die Idee zählt an dieser Stelle.

[4] Ein dynamisches Programm ist im Endeffekt eine Datenstruktur, in der man sich optimale Teillösungen speichert und dann die Einträge der Datenstruktur basierend auf vorher bereits berechneten Einträgen ausfüllt. Der Name „Programm“ ist dabei historisch bedingt.

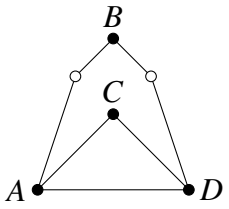


Abbildung 3.3: Beispielgraph

Ferner benötigen wir eine Funktion abb : $E \times E \mapsto \{0,1\}$, die genau dann 1 ist, wenn man zwischen zwei Kanten abbiegen muss. Wie diese berechnet werden kann, wurde bereits besprochen. Für Pfade sei der Wert von abb die Anzahl aller Abbiegungen entlang des Pfades. Dies kann durch Iteration über den Pfad bestimmt werden.

Als Eingabe bekommt unser Algorithmus nun einen solchen Graphen, einen Startknoten s, einen Endknoten t und ein Limit l für die maximale Anzahl an Abbiegungen. Als Ausgabe soll der Algorithmus einen Pfad Q von s nach t ausgeben, der unter allen Pfaden mit $\text{abb}(Q) \leq l$ den Wert $w(Q)$ minimiert.

Algorithmusidee Das dynamische Programm soll ein 2-dimensionales Array A mit den Dimensionen $|V| \times (l+1)$ sein. Nun müssen wir den Daten noch eine Bedeutung geben. Sei zuerst $\mathscr{P}(u,v,k)$ die Menge aller Pfade R von u nach v mit $\text{abb}(R) \leq k$.

Betrachten wir zum Verständnis den Graphen, der in Abbildung 3.3 abgebildet ist. In der Menge $\mathscr{P}(A,D,2)$ sind in diesem Fall 3 Pfade enthalten. Zum einen der direkte Pfad von A nach D. Dieser hat keine Abbiegung und ist deswegen zulässig. Genauso ist der Pfad über C in der Menge enthalten. Zu guter Letzt ist noch der Pfad von A zu D zurück zu A und dann wieder zu D enthalten. Dieser Pfad hat genau 2 Abbiegungen und da wir die Menge $\mathscr{P}(A,D,2)$ nicht weiter eingeschränkt haben ist auch dieser Teil der Menge. In der Menge sind keine Pfade über B enthalten, da diese stets mindestens 3 Abbiegungen benötigen.

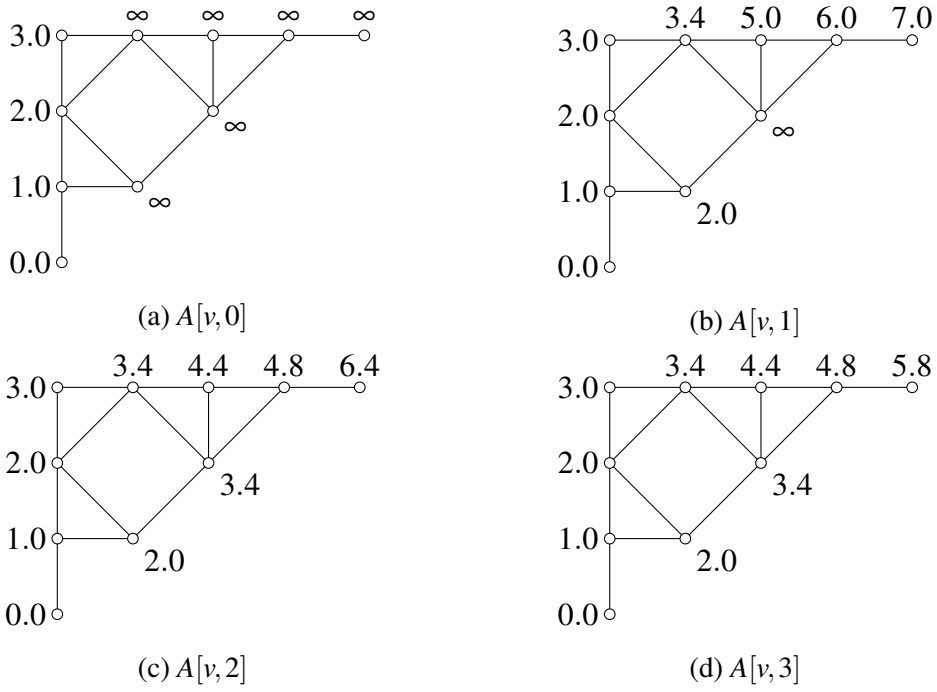
Sei nun also

$$A[v,k] = \begin{cases} \min_{R \in \mathscr{P}(s,v,k)} w(R) & \text{, falls } \mathscr{P}(s,v,k) \neq \emptyset \\ \infty & \text{, sonst} \end{cases}. \tag{3.5}$$

In A steht also für jeden Knoten und jede Abbiegungszahl kleiner gleich l jeweils, ob es einen Pfad von s zu diesem Knoten mit maximal der gegebenen Zahl an Abbiegungen gibt. Falls dem so ist, finden wir auch noch gleich die Länge des kürzesten solchen Pfades. Für den Beispielgraphen findet man in Abbildung 3.4 die Werte zu den Knoten.

Die Länge des Ausgabepfades findet man dann in $A[t,l]$. Jetzt stellen sich aber noch zwei Fragen. Zum einen „Wie kann man diese Datenstruktur schnell berechnen?“ und zum anderen „Was bringt mir die Länge des Pfades, wenn ich doch den Pfad bestimmen soll?“. Diese werden wir nun nacheinander besprechen.

Berechnungsvorschrift Dynamische Programme leben davon, dass die Berechnung der einzelnen Einträge von den bereits ausgerechneten Einträgen abhängt. Da aber nicht alle Einträge von vorher berechneten Werten abhängen können, müssen einige Startwerte gesetzt werden.



(a) $A[v,0]$ (b) $A[v,1]$

(c) $A[v,2]$ (d) $A[v,3]$

Abbildung 3.4: Beispielwerte des dynamischen Programmes

Zuvor sei aber bemerkt, dass für alle $u,v \in V$ alle Pfade in $\mathscr{P}(u,v,0)$ stets die gleiche Länge haben, da keine Abbiegungen erlaubt sind und sie somit nur die euklidische Distanz als Länge haben können. Deshalb können wir eine Funktion $\text{dist}_{\text{ger}} : V \times V \mapsto \mathbb{R}_0^+ \cup \{\infty\}$, welche die Distanz zwischen zwei Knoten u und v ohne Abbiegen angibt, mit

$$\text{dist}_{\text{ger}}(u,v) = \begin{cases} w(\mathscr{P}(u,v,0)) & \text{, falls } \mathscr{P}(u,v,0) \neq \emptyset \\ \infty & \text{, sonst} \end{cases} \tag{3.6}$$

definieren.

Nun setzen wir

$$A[v,0] = \text{dist}_{\text{ger}}(s,v). \tag{3.7}$$

Damit haben wir den Grundstein für die weitere Berechnung gelegt. Für alle $k \in [1,l]$ setzen wir die verbleibenden Werte auf

$$A[v,k] = \min_{u \in V} A[u,k-1] + \text{dist}_{\text{ger}}(u,v). \tag{3.8}$$

Nun bleibt also noch zu zeigen, dass diese Berechnungsvorschrift korrekt ist, und zu besprechen, wie man dist_{ger} effizient berechnen kann. Wenden wir uns also zuerst der Korrektheit zu. Dafür werden wir mathematische Induktion über den ersten Index des dynamischen Programmes verwenden. Wir werden also zuerst zeigen, dass die Initialisierung korrekt ist, und dann, davon ausgehend, dass auch die darauf aufbauenden Berechnungen korrekt sind.

Korrektheitsbeweis Wie oben bemerkt haben für alle Knoten $v \in V$ alle Pfade in $\mathscr{P}(s,v,0)$ stets die gleiche Länge. Folglich ist $\text{dist}_{\text{ger}}(s,v)$ nach Definition genau der Wert, der in $A[0,v]$ stehen muss. Dieser ist damit korrekt berechnet.

Sei nun ein $k \in [1,l]$ gegeben und außerdem nehmen wir an, dass alle Werte von A, bei denen der erste Index echt kleiner als k ist, korrekt berechnet wurden. Nun wollen wir für alle $v \in V$ zeigen, dass der Wert $A[k,v]$ korrekt berechnet wird. Sei also nun O ein kürzester Pfad von s nach v unter den Pfaden mit $\text{abb}(O) \leq k$. Den Fall, dass kein O existiert, handeln wir später ab. Es gilt nun zu zeigen, dass $w(O) = A[k,v]$ ist. Das werden wir beweisen, indem wir zuerst $A[k,v] \leq w(O)$ und danach $A[k,v] \not< w(O)$ beweisen.

($\leq$) Falls $\text{abb}(O) = 0$ gilt, so ist nach Annahme auch $A[k-1,v] = w(O)$. Sei ferner ε der leere Pfad. Dann gilt $w(\varepsilon) = 0$ und $\varepsilon \in \mathscr{P}(v,v,0)$. Also gilt $A[k-1,v] + \text{dist}_{\text{ger}}(v,v) = w(O)$, womit dieser Fall gezeigt ist.

Ansonsten kann man O an der letzten Abbiegung teilen und erhält dann zwei Pfade R und S mit $\text{abb}(R) = \text{abb}(O) - 1$ und $\text{abb}(S) = 0$ als auch $R \diamond S = O$[5]. Sei also nun u der letzte Knoten von R und der erste Knoten von U. Dann gilt $w(R) = A[k-1,u]$, da $R \in \mathscr{P}(s,u,k-1)$ ist und damit nach Annahme $A[k-1,v] \leq w(R)$ gilt. Angenommen, es wäre $A[k-1,v] < w(R)$, dann gäbe es $R' \in \mathscr{P}(s,u,k-1)$ mit $w(R') < w(R)$. Dann würde aber $R' \diamond S \in \mathscr{P}(s,v,k)$ als auch $w(R' \diamond S) = w(R') + w(S) < w(R) + w(S) = w(O)$ gelten. Damit wäre O nicht optimal, was aber der Definition von O widerspricht. Folglich kann dieser Fall nicht eintreten.

Außerdem gilt nach Definition $\text{dist}_{\text{ger}}(u,v) = w(S)$, womit $A[k-1,u] + \text{dist}_{\text{ger}}(u,v) = w(R) + w(S) = w(O)$ ist. Demnach ist auch dieser Fall gezeigt.

($\not<$) Nehmen wir nun an, dass $A[k,v] < w(O)$ gilt. Sei nun $u \in V$ der Knoten, bei dem der Wert von $A[k,v]$ erreicht wurde. Da $w(O)$ und damit auch $A[k,v]$ endlich sind, gilt dies auch für $\text{dist}_{\text{ger}}(u,v)$. Also gibt es einen Pfad mit $S \in \mathscr{P}(u,v,0)$. Ferner ist auch $A[k-1,u]$ endlich und, da dieser Wert nach Annahme korrekt berechnet wurde, ist $\mathscr{P}(s,u,k-1)$ nicht leer. Sei R ein kürzester Pfad aus $\mathscr{P}(s,u,k-1)$. Dann ist $w(R) = A[k-1,u]$ und demnach $w(R) + w(S) = A[k-1,u] + \text{dist}_{\text{ger}}(u,v) < w(O)$.

Außerdem ist $\text{abb}(R \diamond S) \leq \text{abb}(R) + \text{abb}(S) + 1 = \text{abb}(R) + 1 = k$, da eine Konkatenation von 2 Pfaden maximal eine Abbiegung hinzufügt und nach Definition $\text{abb}(S) = 0$ gilt. Demnach ist $R \diamond S \in \mathscr{P}(s,v,k)$. Dies widerspricht aber der Optimalität von O. Da nur logische Schlüsse gezogen wurden, muss damit die Annahme, dass $A[k,v] < w(O)$ gilt, falsch gewesen sein und die Behauptung $A[k,v] \not< w(O)$ gelten.

Sollte es nun kein O geben, die Menge $\mathscr{P}(s,v,k)$ also leer sein, aber $A[k,v]$ endlich sein, so kann man wie im Fall ($\not<$) die Zwischenergebnisse nutzen um einen Pfad in $\mathscr{P}(s,v,k)$ zu konstruieren. Dieser widerspricht dann aber $\mathscr{P}(s,v,k) = \emptyset$. Folglich muss $A[k,v] = \infty$ gelten. Demnach wird der Wert $A[k,v]$ also auch korrekt berechnet.

Damit wurde bewiesen, dass die angegebene Berechnungsvorschrift alle Werte von A korrekt berechnet.

Berechnungen abbiegungsfreier Distanzen Nun bleibt für diesen Abschnitt noch die Berechnung von dist_{ger} zu besprechen. Der Alg. 1 berechnet für einen gegeben Knoten $v \in V$ und alle Knoten $u \in V$ den Wert $\text{dist}_{\text{ger}}(v,u)$. Diesen Algorithmus führen wir dann für alle v aus. Danach haben wir also alle Werte von dist_{ger} vorberechnet und brauchen diese bei der Berechnung des dynamischen Programmes nur noch in einem Array nachschauen. Ferner soll es in

[5] Der Operator $\diamond$ steht dabei für die Konkatenation zweier Pfade.

dem Graphen[6] keine Kanten geben, die an dem selben Knoten starten und beide in die exakt gleiche Richtung gehen.

Algorithmus 1: Berechnung von dist_{ger}

Input: $v \in V$

```
1  for u ∈ V do
2      dist_ger(v,u) ← ∞;
3  dist_ger(v,v) ← 0;
4  for e = {u,v} ∈ E do
5      dist_ger(v,u) ← w(e);
6      cur_vertex ← u;
7      cur_edge ← e;
8      while ∃e' = {cur_vertex, w} ∈ E : abb(cur_edge, e') = 0 do
9          dist_ger(v,w) ← dist_ger(v, cur_vertex) + w(e');
10         cur_vertex ← w;
11         cur_edge ← e';
```

Nun gilt es noch die Korrektheit des beschriebenen Algorithmus' zu beweisen und seine Laufzeit zu diskutieren. Zuerst werden wir über Induktion zeigen, dass für alle Knoten $w \in V$ mit $\text{dist}_{\text{ger}}(v,w) < \infty$ der korrekte Wert gesetzt wird. Bezeichne dazu $\text{dist}_{\text{ger}_A}$ die Ausgabe des Algorihtmus. Für alle Knoten, die inzident zu v, oder v selbst sind, wird offensichtlich dist_{ger} korrekt gesetzt.

Sei nun $k \in \mathbb{N}_{>1}$ gegeben und sei $w \in V$ ein Knoten so, dass die Anzahl der Knoten im Pfad mit der geringsten Knotenzahl in $\mathscr{P}(v,w,0)$ genau k ist. Sei ferner die Behauptung für alle Knoten, für die dieser Wert echt kleiner k ist korrekt. Sei nun $R = (v,\ldots,u,w) \in \mathscr{P}(v,w,0)$ der angesprochene Pfad. Dann wurde nach Annahme $\text{dist}_{\text{ger}}(v,u)$ korrekt berechnet. Sei nun e die Kante zu u in R. Dann muss $\text{abb}(e,\{u,w\}) = 0$ gelten. Außerdem kann es keine weitere Kante e' mit $\text{abb}(e,e') = 0$ geben, da sich die Kanten e' und $\{u,w\}$ sonst überlagern müssten. Demnach ist $\text{dist}_{\text{ger}_A}(v,w) = \text{dist}_{\text{ger}_A}(v,u) + w(e)$. Sei nun R' der Pfad R ohne den letzten Knoten. Dann ist nach Induktionsannahme $w(R') = \text{dist}_{\text{ger}_A}(v,u)$. Folglich ist $\text{dist}_{\text{ger}_A}(v,w) = w(R') + w(e) = w(R)$ und damit gleich $\text{dist}_{\text{ger}}(v,w)$. Da $\text{dist}_{\text{ger}_A}(v,w)$ danach nicht neu gesetzt wird, wird der Wert damit korrekt berechnet.

Jetzt gilt es also zu zeigen, dass auch alle Knoten $u \in V$ mit $\text{dist}_{\text{ger}}(v,u) = \infty$ korrekt berechnet werden. Nehmen wir also an, es gäbe einen solchen Knoten für den $\text{dist}_{\text{ger}_A}(v,u) \neq \infty$ gilt. Dann müsste dieser Wert in Zeile 3,7 oder 9 gesetzt wurden sein. Aber sollte dies der Fall sein, so kann man die bisherigen `cur_edges` zurückgehen und so einen Pfad R mit $\text{abb}(R) = 0$ beginnend bei v und endend bei u finden. Somit wäre aber $\text{dist}_{\text{ger}}(v,u) \neq \infty$, was aber angenommen wurde. Folglich ist auch dieser Fall korrekt berechnet.

Laufzeitbetrachtung Nachdem wir nun bewiesen haben, dass alle genutzten Algorithmen korrekt sind, besprechen wir nun die Laufzeit genauer. Als Eingabe bekommen wir einen Graphen mit n Knoten und m Kanten, sowie eine natrüliche Zahl l, die die maximale Anzahl an Abbiegungen angibt.

[6] besser gesagt, der euklidischen Einbettung des Graphen

Der Algorithmus zum Berechnen von dist_{ger} benötigt $O(n)$ zur Initialisierung. Zu jedem Knoten ist der Pfad ohne Abbiegungen eindeutig bestimmt, da alle Knoten auf diesem auch auf der Geraden, die durch v und den anderen Knoten verläuft liegen müssen. Für v selbst ist der Pfad auch eindeutig bestimmt, denn er muss leer sein.

Da sich Kanten auch nicht überlagern können, gibt es an jedem Zwischenknoten genau eine Kante, die zulässig für den nächsten Schleifendurchlauf ist. Zu den Knoten, die in der unteren Schleife abgearbeitet werden, kann man jeweils den Pfad zu v rekonstruieren. Da für den gleichen Pfad jeder Knoten nur einmal bearbeitet wird, kann jeder Knoten in der unteren Schleife maximal einmal abgearbeitet werden.

Als Preprocessing können wir die Kanten an jedem Knoten nach ihrem Richtungsvektor sortieren[7]. Da zu jeder Kante e die Kante auf die man ohne Abbiegung kommt den gleichen Richtungsvektor hat, muss sie in der sortierten Adjazenzliste neben e liegen. Demnach kann die Überprüfung der Schleife in $O(1)$ erfolgen. Damit wird für jeden Knoten in der unteren Schleife maximal $O(1)$ Arbeit verrichtet. Demnach wird über alle Kanten in der unteren Schleife nur $O(n)$ Arbeit verrichtet.

Ruft man Alg. 1 für alle Knoten auf, so wird $O(n \cdot (n + m \log n))$ Arbeit verrichtet. Die Kanten müssen aber nicht jedes Mal neu sortiert werden. Also ist ein Aufwand von $O(n^2 + m \log n)$ zur Berechnung von allen Werten von dist_{ger} ausreichend.

Um das dynamische Programm zu berechnen, muss zur Initialisierung $O(n)$ Arbeit verrichtet werden. Für jede Zelle danach wird $O(n)$ zusätzlicher Aufwand fällig. Es gibt $O(n \cdot l)$ viele Zellen. Also benötigen wir $O(n^2 \cdot l)$ Zeit um das gesamte dynamische Programm zu füllen.

Die Gesamtzeit des Algorithmus beträgt damit $O(n^2 \cdot l + m \log n)$ Laufzeit.

Pfadrekonstruktion Nun betrachten wir noch, was gemacht werden muss, um den Pfad zusätzlich zu der Länge zu bestimmen. Dafür lohnt es sich zu überlegen, wo die Information verloren geht.

Zuerst fällt auf, dass bei der Bestimmung von dist_{ger} stets der Pfad verwurfen wird. Dies lässt sich aber leicht lösen, indem man beim jedem Setzen von dist_{ger} in der unteren Schleife zusätzlich zu den jeweiligen Wert von `cur_vertex` speichert. Für v setzen wir einen gesonderten Wert, der das Ende des Pfades symbolisiert. Nun können wir die Werte von `cur_vertex` verfolgen. Dabei erhalten wir also eine verkettete Liste, in der der Pfad gespeichert wird.

Dabei ist es wichtig die Eigenschaft von verketteten Listen zu nutzen, dass Teile von mehreren Listen geteilt werden. Sollten wir beispielsweise jeweils einen Vektor pro Knoten speichern, so muss zur Abarbeitung eines Knoten der vorgegangene Pfad kopiert werden. Dadurch steigt die asymptotische Laufzeit, in der ein Knoten abgearbeitet werden kann, auf $O(n)$. Damit würde die Gesamtlaufzeit auf $O(n^3)$ steigen. Mit unserer Lösung wird die asymptotische Komplexität nicht beeinflusst.

Um zu zeigen, dass diese Liste den korrekten Pfad beschreibt, kann man den Beweis zur Korrektheit der Länge erweitern. In der Induktion wissen wir, dass der Pfad zum Vergänger korrekt ist. Wir wissen außerdem, dass der Pfad durch die betrachtete Kante korrekt erweitert werden kann und dass `cur_vertex` zum entsprechenden Zeitpunkt der Vorgängerknoten ist. Also ist der Pfad auch zum aktuellen Knoten korrekt.

[7] Diese Sortierung kann beispielsweise lexikographisch vorgenommen werden.

Wir haben nun bei der Berechnung von dist_{ger} für jede mögliche Eingabe auch den entsprechenden Pfad gefunden. Nun müssen wir noch in dem dynamischen Programm die Pfade mit speichern. Dabei bedienen wir uns der gleichen Idee und speichern uns zusätzlich zur Länge noch den Knoten u mit, der in Gleichung (3.8) zu dem Minimum führt. Nun können wir von der Zelle $A[l,t]$ die Knoten u rückwärts verfolgen und die zu dem entsprechenden dist_{ger} gespeicherten Pfade konkatenieren. Sollte $A[l,t] = \infty$ sein, so wissen wir, dass kein Pfad existiert, und können dies auch zurück geben.

Das Korrektheitsargument für diese Erweiterung ist bereits in dem Fall $(\not<)$ versteckt. Dort wurde nämlich genau der Pfad, den wir uns jetzt abspeichern, aus den Längewerten konstruiert und gezeigt, dass dieser zulässig ist.

3.1.3 Verknüpfung der Ergebnisse

Nun wollen wir das eigentliche Problem lösen. Gegeben eine Längenbeschränkung, finde einen Pfad, der höchstens die gegebene Länge hat und unter diesen Pfaden die Anzahl an Abbiegungen minimiert.

Betrachten wir dafür zuerst einen solchen Pfad R und den Fall, dass es einen Knoten u gibt, der mehrmals in R vorkommt. In dem Pfadsegment zwischen dem ersten und letzten Vorkommen von u muss mindestens eine Abbiegung gewesen sein. Wenn wir dieses Segment entfernen, fügen wir aber maximal eine Abbiegung hinzu. Also erhöht man dadurch die Zahl der Abbiegungen nicht. Demnach gibt es also stets einen zulässigen Pfad, bei dem dieser Knoten maximal einmal vorkommt.

Diese Argument kann man für jeden mehrfach vorkommenden Knoten anführen. Deshalb gibt es auch einen zulässigen Pfad, in dem jeder Knoten maximal einmal vorkommt. Da am ersten und letzten Knoten keine Abbiegungen sind, muss $\text{abb}(R) \leq n-2$ gelten.

Berechnen wir nun das dynamische Programm für diesen Graphen und $l = n-2$. Wir können nun also das kleinste k so finden, dass $A[k,t]$ nicht größer als die gewünschte Länge ist. Es kann nun nach Definition keinen Pfad mit weniger als k Abbiegungen von s nach t geben, der auch kurz genug ist. Der Pfad, den wir durch Rückverfolgung der gespeicherten Werte finden, ist damit optimal.

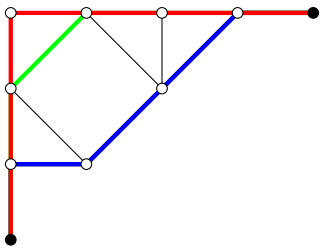
Die gleiche Konstruktion, die wir verwendet haben, um zu zeigen, dass jeder Knoten in einem Pfad mit minimal vielen Abbiegungen maximal einmal vorkommt, kann auch genutzt werden um zu zeigen, dass in einem kürzesten Pfad jeder Knoten maximal einmal vorkommt. Also hat auch der kürzeste Pfad maximal $n-2$ Abbiegungen. Also ist $A[n-2,t]$ die Länge des kürzesten Pfades von s nach t. Wir benötigen also um das ursprüngliche Problem zu lösen nicht einmal einen zusätzlichen kürzeste Wege Algorithmus zu nutzen.

Gegeben einen Faktor $f \in \mathbb{R}_{\geq 1}$ können wir also das kleinste k so suchen, dass $f \cdot A[k,t] \leq A[n-2,t]$ gilt. Damit haben wir die Aufgabe mit einer Gesamtlaufzeit von $O(n^2 \cdot (n-2) + m \log n) = O(n^3)$ gelöst.

3.2 Beispiele

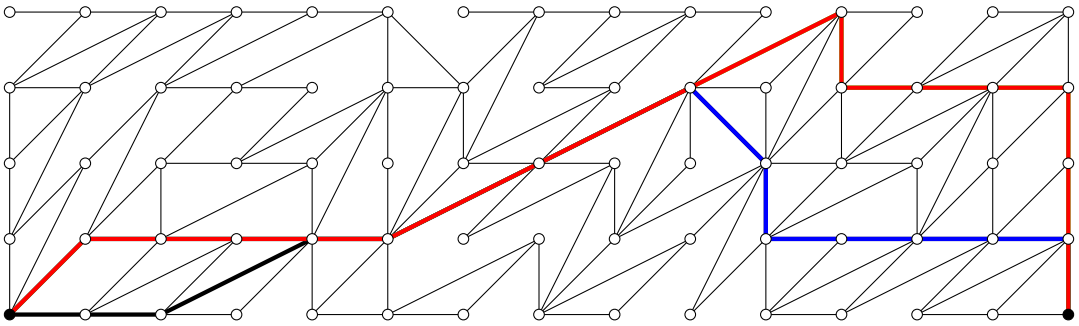
Die Beispiele sind in den folgenden Absätzen visualisiert. Dazu sei bemerkt, dass stets alle Pfade gezeichnet sind, aber die Pfade mit einem größerem erlaubten Faktor die anderen Pfade überlagern. Darüber hinaus sind die Start- und Endpunkte in Schwarz markiert.

abbiegen0.txt:



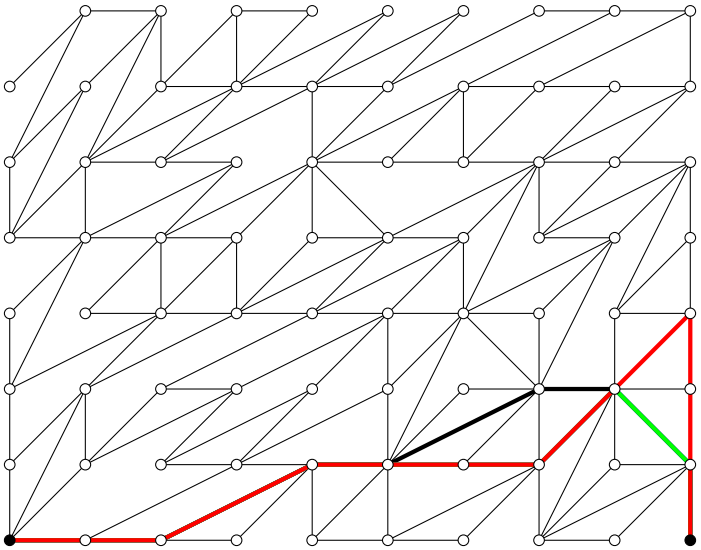
- +30 % $\mapsto$ 1 Abbiegungen mit Länge 7.000
- +15 % $\mapsto$ 2 Abbiegungen mit Länge 6.414
- +10 % $\mapsto$ 3 Abbiegungen mit Länge 5.828
- +00 % $\mapsto$ 3 Abbiegungen mit Länge 5.828

abbiegen1.txt:



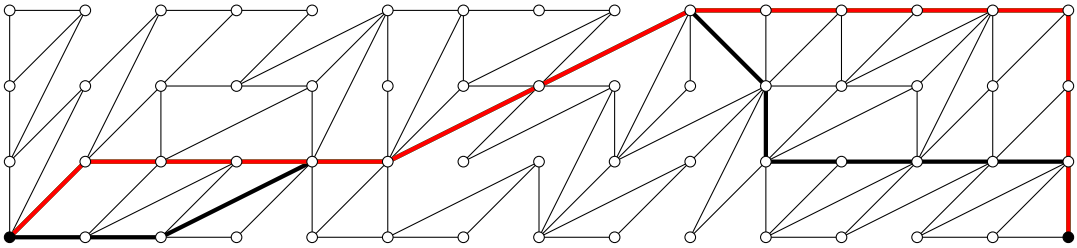
- +30 % $\mapsto$ 5 Abbiegungen mit Länge 19.122
- +15 % $\mapsto$ 5 Abbiegungen mit Länge 19.122
- +10 % $\mapsto$ 6 Abbiegungen mit Länge 17.301
- +00 % $\mapsto$ 7 Abbiegungen mit Länge 17.122

abbiegen2.txt:



- +30 % $\mapsto$ 4 Abbiegungen mit Länge 13.064
- +15 % $\mapsto$ 5 Abbiegungen mit Länge 11.064
- +10 % $\mapsto$ 5 Abbiegungen mit Länge 11.064
- +00 % $\mapsto$ 6 Abbiegungen mit Länge 10.886

abbiegen3.txt:

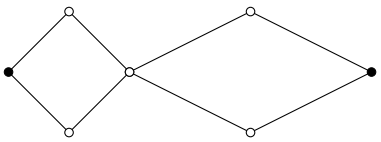


- +30 % $\mapsto$ 4 Abbiegungen mit Länge 17.886
- +15 % $\mapsto$ 4 Abbiegungen mit Länge 17.886
- +10 % $\mapsto$ 4 Abbiegungen mit Länge 17.886
- +00 % $\mapsto$ 7 Abbiegungen mit Länge 17.122

3.3 Mögliche Erweiterungen

- Abbiegungen können gewichtet werden. Beispielsweise könnte man für eine Abbiegung variierende Kosten einführen. Dann kann man einen Weg finden, der diese Kosten als Schranke bekommt.
- Straßen müssen nicht gerade verlaufen. Dann bietet sich an, als Definition des Geradeausfahrens die Tangenten in dem Übergangspunkt zu betrachten. Natrürlich gehört hier eine sinnvolle Berechnung der Länge dieser Straßenabschnitte dazu.

- Es könnten alle Pfade, die den Ansprüchen von Bilal genügen, gefunden werden – also die Pfade mit der geringsten Abbiegungszahl, deren Länge einen bestimmten Schwellwert nicht überschreitet. Hierbei sollte aber darauf eingegangen werden, dass die Laufzeit des Algorithmus dadurch exponentiell wird.



In dem dargestellten Graphen haben alle der vier kürzesten Pfade drei Abbiegungen. Wenn man diesen Graph noch einmal mit sich selbst erweitert, hat man dann 16 Möglichkeiten für den kürzesten Pfad. Im Allgemeinem vervierfacht sich die Anzahl für jedes Mal Erweitern.

Da auch alle kürzesten Pfade die geringste Anzahl an Abbiegungen aufweisen, müsste man Bilal eine exponentielle Anzahl an Pfaden vorschlagen. Dies führt direkt zu einer exponentiellen Laufzeit. Dieses Problem kann man wegen der Ausgabegröße auch nicht subexponentiell lösen.

- Es könnten andere Optimierungskriterien, wie zum Beispiel die Gesamtzahl an Kreuzungen, an denen Bilal geradeaus fahren kann, betrachtet und optimiert werden. Diese lassen sich, soweit sie diskret sind, oft gut in das dynamische Programm einbauen.

3.4 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Karte adäquat modelliert*: Die Karte wurde geeignet modelliert, zum Beispiel als Graph. Falls nötig, sollte darauf eingegangen werden, wie die Punkte in der Ebene als Knoten modelliert werden.

(2) *Abbiegen sinnvoll definiert*
   - Es wurde definiert, was eine Abbiegung ist. Die Definition ist angemessen und wurde bei der Bearbeitung korrekt berücksichtigt. Die Überführung in eine Normalform, wie in der Beispiellösung, ist dabei nicht nötig.
   - Senkrechte Straßen müssen korrekt behandelt werden, insbesondere wenn Abbiegungen über Steigungsunterschiede bestimmt werden.
   - Ob der Übergang von einer Kante zu einer anderen eine Abbiegung darstellt, kann auch in einer geeigneten Datenstruktur gespeichert werden, etwa in einem erweiterten Graphen.

(3) *Laufzeit des Verfahrens in Ordnung*: Da ein Verfahren mit polynomieller Laufzeit existiert, wird auch ein solches erwartet. Exponentielle Laufzeiten führen zu Punktabzug. Sie können höchstens durch Erweiterungen gerechtfertigt sein.

(4) *Verfahren mit optimalen Ergebnissen*
   - Es werden optimale Lösungen erwartet. Die Abbiegungen der ausgegebenen Pfade sind also für den gegebenen Verlängerungsfaktor minimal.

- Es ist sinnvoll, unter den Pfaden mit minimaler Abbiegezahl (im Rahmen der erlaubten Verlängerung) einen kürzesten zu empfehlen. Das ist aber nicht gefordert.

(5) *Verfahren liefert korrekte Ergebnisse*: Insbesondere Weglängen und die Verhältnisse zwischen Weglängen werden grundsätzlich korrekt berechnet.

2. Theoretische Analyse

(1) *Verfahren / Qualität ingesamt gut begründet*: Der Verlängerungsfaktor bezieht sich auf einen insgesamt kürzesten Weg von Start zu Ziel. Es muss also auch (kurz) beschrieben sein, wie dieser insgesamt kürzeste Weg berechnet wird.

(2) *Gute Überlegungen zur Laufzeit:* Hier sollte insbesondere richtig eingeschätzt werden, ob bzw. dass das gewählte Verfahren eine polynomielle Laufzeit hat.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Die geforderten Ergebnisse ($+10\%$, $+15\%$, $+30\%$) sind für alle Beispiele (mindestens 1 bis 3) dokumentiert. Es ist also akzeptabel, wenn die Ergebnisse für Beispiel 0 (aus der Aufgabenstellung) fehlen. Häufig wurden Angaben für $+10\%$ weggelassen, vermutlich weil in der Aufgabenstellung von diesem Wert nicht die Rede ist; ausnahmsweise ist auch das akzeptabel.

(5) *Ergebnisse nachvollziehbar dargestellt*: Auch bei dieser Aufgabe ist eine grafische Darstellung der Ergebnisse naheliegend. Dabei ist allerdings wichtig, dass die Unterschiede zwischen den empfohlenen Wegen gut erkennbar sind. Aber auch ohne eine grafische Darstellung können die Ergebnisse gut nachvollzogen werden; zumindest müssen aber für alle Wege die Längen angegeben sein.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

- Das Programm verbraucht weniger als Chrome mit einem Tab. *– (Weniger was?)*
- Diese lässt sich nach unserem Wissen nicht weiter verbessern, was nicht bedeutet, dass es nicht noch eine bessere und effizientere Möglichkeit gibt.
- Ich möchte mich hiermit aus ganzem Herzen für diese äußerste Überlänge entschuldigen. Ich weiß, dass Sie als Bewerter an diesem Wochenende sehr viele Arbeiten bewerten müssen und ich hatte wirklich nicht vor, dass meine Arbeit einen so großen Teil ihrer Zeit einnehmen sollte. *– (Es folgten tatsächlich 18 (achtzehn!) ACHTZEHN Seiten mathematischer Beweis für das Programm.)*
- Beim Berechnen gibt das Programm kleine Texte wie „Gleich fertig“ aus. Diese sind zufällig aus einer Liste gewählt. *– ([„Bitte kurz warten“, „Gleich fertig“, „Einen Moment bitte“, „Bitte haben sie etwas Geduld“, „Nicht mehr lange“])*
- Auch wenn diese Klassen auf den Materialien zu den zentralen NRW-Abiturprüfungen basieren, wurden diese von mir grundlegend überarbeitet.
- Falls noch Fragen bezüglich des Dijkstra-Algorithmus verbleiben verweise ich auf die Literatur im Internet. Hier ist dieses Video sehr empfehlenswert :) `https://www.youtube.com/watch?v=2poq1Pt32oE&t=327s` *– („Literatur“)*
- Zur Ermittlung der Laufzeiten werden diese im Programmablauf bestimmt. Hiermit können die Laufzeitoptimierungen durch die Heuristiken praktisch bewiesen werden.
- *Diverse Typos:* räpresentieren, Algorythmus *(Klassiker)*, optimalsten, Dijakstras Algorithmus, Branch-and-Bond Algorithmus, Lauzeitabschäzung, `bool isWohleNumber()`, expondentiellen Wachtum, sukkessiv

Aufgabe 1:

- Mein Lösungsansatz für das Berechnen eines Stromrallyespiels beruht, soweit ich dies beurteilen kann, auf dem Prinzip des Backtrackings.
- Das Programm ist zwar nicht schnell, aber dafür auch nicht korrekt.
- Dijkstra-ähnlicher Algorithmus (Fußnote: Der Algorithmus ist dem Dijkstra-Algorithmus ähnlich, nicht Edsger W. Dijkstra.)
- Ich habe die Lösungsidee in drei Teile eingeteilt, da die Aufgabe aus zwei Teilen besteht.
- Die Funktion `hoechsteEntfernungZuEinerAnderenBatterie` sucht in einer Schleife den niedrigsten Abstand zu einer anderen Batterie.
- ```
  public static void main(String[] args){
      <ganz viel auskommentierter Code>
      while (true) {System.out.println("berechne...");}
  }
  ```
  ```

Aufgabe 2:

- Leider musste ich das Programm abgeben, bevor die Sonne ausglüht, und so sah ich mich gezwungen Optimierungen anzuwenden.
- Die maximale Fakultät 11 resultiert daraus, dass sie die letzte Zahl ist, die man Fakultieren kann, ohne den Bereich des Integer zu überschreiten.
- Die praktische Laufzeit des Programms erhöht sich durch diese beiden Datenstrukturen enorm. *– (Wieso benutzt du sie dann? :D)*
- Dass Fehler in den Ergebnissen auftreten, liegt vollständig an der Implementierung und der Unfähigkeit des Programmierers.

Aufgabe 3:

- Der Algorithmus beginnt damit, alle linearen Vertices zu triangulären zu machen.
- Da ich den Weg mit einer Breitensuche durchgehe, kann man sich nicht verhängen, ohne weiter zu kommen.
- Es gibt vier Richtungen, oben und unten, rechts und links, schräg oben recht und unten links, sowie schräg unten rechts und oben links.
- `laengederlisteVonListenVonKoordinaten`
- Alternativwege für die Alternativwege der Alternativwege
- Je nachdem, welcher der beiden Werte größer, kleiner oder gleich null ist, wird ein anderer Rückgabewert ausgegeben. Darunter die booleschen Werte True, False und None, sowie einige beliebig gewählte Fehlernamen, um die Richtungen zu unterscheiden. *– (Im Code mussten wir sehen, dass das wirklich gemacht wurde . . .)*
- Im ersten Teil der Aufgabe ist im Prinzip die Laufzeit einer Breitensuche. Dementsprechend skaliert sie mit $O(n^m)$, wobei n die Anzahl der Knoten ist und m die Anzahl an Kanten.
- Bei der `closedList` handelt es sich um ein `Set [...]`. Bei der `closedList` handelt es sich um eine eigene Klasse `SortedList`, die von der `ArrayList` erbt. *– (Was denn nun?)*
- Bilal hat zusätzlich Ambitionen entwickelt, über die Erde hinaus zu radeln und per Anhalter durch die Galaxis zu reisen. Er leidet aber immer noch unter starker Deflexiophobie und will auf seinen Reisen möglichst selten abrupt seine Richtung ändern.
- Bevor ich zur eigentlichen Lösungsidee komme, müssen ein paar Begriffe erklärt werden, die ich im Laufe dieser Dokumentation verwende: „der kürzeste Weg“ – bezeichnet den kürzesten Weg für das jeweilige Straßennetz. Es wird davon ausgegangen, dass man sich sicher sein kann, dass dieser berechnete Weg wirklich der kürzeste ist.
- Vernachlässigt man die schlechte Laufzeit meiner Hilfsfunktionen, hat mein Algorithmus im worst-case die gleiche Laufzeit wie A* im worst-case.

- Zu Bilal und seinem Freund: (Aufgrund der momentanen Kontaktbeschränkungen sollten sie sich beide sowieso eigentlich nicht treffen.)
- Diese Lösungsmethode heißt Brute-Force-Methode (Methode der rohen Gewalt) und ist in der Informatik verbreitet
- Auch wenn man hier Ergebnisse zwischenspeichern könnte, um Berechnungszeit zu sparen, wurde dies nicht implementiert, da die Laufzeit bereits ohne diese Operation schnell genug ist.