Aufgabe 3: Hex-Max

Teilnahme-ID: 63421

Bearbeiter/-in dieser Aufgabe:
Niels Glodny

25. April 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 Modellierung

Gegeben ist ein Wort w aus den Hex-Ziffern $x_1, x_2, ..., x_n$, die jeweils eine Stelle der Hexadezimalzahl beschreiben. Jedes Zeichen ist ein 7-Tupel, das für jedes Segment im Zeichen angibt, ob es belegt oder frei ist. Gesucht ist nun das größte solche Wort, was sich mit maximal m Umlegungen aus w legen lässt.

1.2 Lösung

Das Problem kann mittels dynamischer Programmierung und einem anschließenden Greedy-Algorithmus gelöst werden. Sei

$$f(p, e) \tag{1}$$

die Anzahl der Umlegungen, die mindestens benötigt werden, um insgesamt e zusätzliche Positionen im Teilwort $w_p, ..., w_n$ zu besetzen und eine valide Zahl zu erhalten. Die Zahl e kann auch negativ sein, was dann bedeutet, dass Stäbchen von e Positionen entfernt werden sollen. Dabei zählt nur das Entfernen von Stäbchen aus dem Teilwort als Umlegung. Ist e positiv, so können diese Stäbchen, ohne als Umlegung zu zählen, abgelegt werden. Ist e negativ, so zählt das Nehmen der e Stäbchen als Umlegung. Diese Konvention wird im Folgenden für alle Umlegungen verwendet. So wird sichergestellt, dass Umlegungen zwischen Zeichen immer nur einmal gezählt werden.

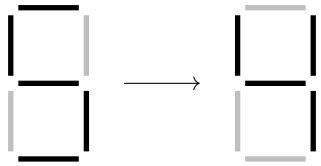


Abbildung 1: Umlegung der Ziffer 5 zur Ziffer 4. $d(5,4) = 1$, da ein Stab umgelegt werden muss. $n(5,4) = -1$, da ein weiterer Stab entfernt werden muss. Der entfernte Stab zählt nicht als Umlegung. Müsste aber ein Stab hinzugefügt werden, so würde dies als Umlegung zählen. $d(4,5)$ wäre also 2.

Ist $f(p,e)$ für alle p und e bekannt, kann das maximale Wort über ein Greedy-Verfahren rekonstruiert werden. $f(p,e)$ kann über dynamische Programmierung effizient berechnet werden.

Sei dazu $d(x,y)$ die Anzahl der Umlegungen, die nötig sind, um die Ziffer x in die Ziffer y umzuwandeln. Für entfernte bzw. hinzugefügte Stäbchen gilt die oben beschriebene Konvention. $n(x,y)$ ist analog dazu die Anzahl der Stäbchen, die entfernt bzw. hinzugefügt werden müssen, um x in y umzuwandeln. Abbildung 1 zeigt ein Beispiel für von $d(x,y)$ und $n(x,y)$. Beide Werte können einfach aus den Segmenten für die Ziffern bestimmt werden. Nun gilt die Rekurrenz

$$f(p,e) = \begin{cases} 0 & \text{, wenn } p > n \text{ und } e = 0 \\ \infty & \text{, wenn } p > n \text{ und } e \neq 0\,. \\ \min_{x\in[0,15]} f(p+1, e+n(w_p,x)) + d(w_p,x) & \text{, andernfalls} \end{cases} \tag{2}$$

Die ersten beiden Fälle geben die Werte für das leere Wort an. Muss kein extra Stäbchen gelegt werden, so ist das leere Wort ein valides Wort. Muss dagegen die Anzahl der Stäbchen verändert werden, so kann kein valides Wort mehr entstehen.

Die aktuelle Stelle p kann zu einer der Hex-Ziffern 0–F umgewandelt werden. Dafür werden $d(w_p,x)$ Umlegungen benötigt. Zusätzlich werden die Umlegungen addiert, die noch benötigt werden, um die $n(w_p,x)$ frei gewordenen Stäbchen in den Rest des Wortes zu integrieren. $f(p,e)$ ist dann der minimale Wert über alle möglichen Hex-Ziffern. Mit der Rekurrenz 2 kann $f(p,e)$ für alle möglichen Werte bottom-up vorberechnet werden, wie in Algorithmus 1 dargestellt. Für e kommen nur Werte $-m \leq e \leq m$ infrage, da andere Lösungen in jedem Fall mehr als m Umlegungen verwenden.

Algorithmus 1 Berechnung von f

```
1: procedure VORBERECHNEN(w, m)
2:     sei dp[1, ..., n + 1][−m, ..., m] ein neues Array
3:     for p ← n, ..., 1 do
4:         for e ← −m, ..., m do
5:             dp[p][e] ← min_{x∈[0,15]} f(p + 1, e + n(w_p, x)) + d(w_p, x)
6:         end for
7:     end for
8:     return dp
9: end procedure
```

Ist $f(p,e)$ bekannt, kann die größte Zahl greedy rekonstruiert werden. Dazu wird an jeder Stelle, von links nach rechts, die größte Ziffer verwendet, bei der die dann benötigten Umlegungen die Grenze noch nicht überschreiten. Dieses Vorgehen ist in Algorithmus 2 dargestellt. Nun ist die größte Zahl gefunden. In der Aufgabenstellung ist jedoch weiterhin gefordert, die Zwischenstände nach jeder Umlegung auszugeben. Dafür muss eine Folge von konkreten Umlegungen gefunden werden, um vom Eingabewort zum Ergebnis zu gelangen. Ferner ist gefordert, dass während der Umlegungen ein Zeichen niemals vollständig geleert werden darf. Die Grundidee dafür ist es, alle Positionen zu finden, in denen ein Stäbchen benötigt bzw. überflüssig ist. Daraufhin muss immer ein Stäbchen von einer überflüssigen zu einer benötigten Position bewegt werden. Da es genau so viele Positionen gibt, in denen ein Stäbchen benötigt wird, wie in denen ein Stäbchen überflüssig ist, ist es prinzipiell egal, welches Stäbchen wohin und in welcher Reihenfolge bewegt wird, vorausgesetzt, es wird immer ein überschüssiges Stäbchen zu einer benötigten Position bewegt. Die Anzahl der Umlegungen ist immer gleich. Allerdings muss darauf geachtet werden, dass keine Stelle zu einem Zeitpunkt vollständig geleert wird. Algorithmus 3 findet greedy die Positionen, an denen ein Stäbchen überflüssig bzw. benötigt ist und fügt diese Positionen zu Listen hinzu. Nach jedem Zeichen

Algorithmus 2 Rekonstruktion der maximalen Zahl

```
1:  procedure REKONSTRUKTION(w, f)
2:      e ← 0                                   ▷ aktuell überschüssige Stäbchen
3:      z ← m                                   ▷ noch übrige Umlegungen
4:      s ← ϵ                                   ▷ das rekonstruierte Wort
5:      for p ← 1, ..., n do
6:          for x ← F, ..., 0 do
7:              if f(p + 1, e + n(w_p, x)) + d(w_p) ≤ z then
8:                  e ← e + n(w_p, x)
9:                  s ← s + x
10:                 z ← z − d(w_p)
11:                 break
12:             end if
13:         end for
14:     end for
15:     return s
16: end procedure
```

Algorithmus 3 Finden der Umlegungen

```
1:  procedure UMLEGEN(w_1, w_2)
2:      e ← []                                  ▷ Positionen mit überschüssigem Stäbchen
3:      n ← []                                  ▷ Positionen, die ein Stäbchen benötigen
4:      for z ← 1, .., n do                     ▷ über alle Zeichen iterieren
5:          for i ← 0, ..., 6 do                ▷ über die sieben Segmente iterieren
6:              if Segment i von Zeichen z muss hinzugefügt werden then
7:                  n.push((z, i))
8:              end if
9:              if Segment i von Zeichen z muss entfernt werden then
10:                 e.push((z, i))
11:             end if
12:         end for
13:         while e.size() > 0 und n.size() > 0 do
14:             a ← e.pop()
15:             b ← n.pop()
16:             bewege Stäbchen von a nach b
17:         end while
18:     end for
19: end procedure
```

wird jeweils ein benötigtes Stäbchen und ein überschüssiges Stäbchen aus den Listen entfernt und das Stäbchen bewegt. Dadurch wird, ohne die Laufzeit von $\Theta(n)$ zu verschlechtern, sichergestellt, dass ein Zeichen niemals vollständig geleert werden kann. Der Beweis dafür folgt unter 1.5.

1.3 Analyse der Laufzeit

Zunächst wird die Laufzeit von Algorithmus 1 analysiert. Die Initialisierung des Arrays in Zeile 2 benötigt $T(n,m) = \Theta(n(2m+1)) = \Theta(nm)$ Laufzeit. Zeile 5 benötigt eine Laufzeit von $\Theta(1)$, da der Ausdruck 15 Mal mit jeweils konstanter Laufzeit ausgewertet wird, unabhängig von n und m. Diese Zeile wird durch die beiden for-Schleifen $n(2m+1)$-Mal ausgeführt. So ergibt sich eine Gesamtlaufzeit von

$$T(n,m) = \Theta(n \cdot m) + n(2m+1) \cdot \Theta(1)$$
$$T(n,m) = \Theta(nm)$$

Nun wird die Laufzeit von Algorithmus 2 analysiert. Die Zeilen 2–4 benötigen insgesamt $T(n) = \Theta(1)$ Laufzeit. Die for-Schleife in Zeile 5 wird n-Mal ausgeführt, die for-Schleife in Zeile 6 wird maximal 15-Mal ausgeführt und die Zeilen 7-12 benötigen insgesamt $T(n,m) = \Theta(1)$ Laufzeit. So ergibt sich für Algorithmus 2 eine Laufzeit von

$$T(n,m) = n \cdot O(15) \cdot \Theta(1)$$
$$T(n,m) = n \cdot \Theta(1)$$
$$T(n,m) = \Theta(n)$$

Algorithmus 3 hat wie bereits erwähnt eine Laufzeit von $T(n) = \Theta(n)$. Die Zeilen 2 und 3 benötigen $\Theta(1)$ Zeit und werden nur einmal ausgeführt. Die for-Schleife aus Zeile 4 wird n-Mal ausgeführt. Die Zeilen 5-12 benötigen pro Ausführung $\Theta(1)$ Laufzeit, da die for-Schleife in Zeile 5 immer 7-Mal ausgeführt wird und jeweils eine konstante Laufzeit pro Iteration hat.

Die while-Schleife in Zeile 13 wird während des gesamten Algorithmus nicht öfter als $7n$-Mal aufgerufen, da jedes Segment nur einmal in die Listen eingefügt (und folglich auch herausgenommen) werden kann. Die Zeilen 13-17 benötigen also über alle Iterationen $O(n)$ Laufzeit. Für die Gesamtlaufzeit von Algorithmus 3 ergibt sich also

$$T(n) = n \cdot \Theta(1) + O(n)$$
$$T(n) = \Theta(n).$$

Zusammen haben die drei Algorithmen eine Laufzeit von

$$T(n,m) = \Theta(n) + \Theta(nm) + \Theta(n)$$
$$T(n,m) = \Theta(nm).$$

Die Laufzeit des Lösungsverfahrens wächst also asymptotisch linear zur Länge des Wortes und zur Maximalzahl der Umlegungen. Da auch der konstante Faktor relativ gering ist, reicht diese Laufzeit in der Praxis auch zum Lösen von sehr großen Instanzen.

1.4 Analyse des Speicherbedarfs

Das *dp*-Array von Algorithmus 1 benötigt insgesamt $n \cdot (2m+1) = \Theta(nm)$ Speicher. Zusammen mit den Variablen p, e und x ergibt sich ein Speicherbedarf von $\Theta(nm) + \Theta(1) = \Theta(nm)$.

Die Rekonstruktion verwendet lediglich die Ganzzahlen e, p, z und x, die einen Speicherbedarf von $\Theta(1)$ haben. Der String s wird genauso lang wie das Eingabewort und benötigt daher $\Theta(n)$ Speichereinheiten. Insgesamt hat Algorithmus 2 also einen Speicherbedarf von $\Theta(1) + \Theta(n) = \Theta(n)$.

Algorithmus 3 verwendet zwei Listen. Da jedes Segment nur maximal einmal in eine der Listen eingefügt wird, können diese zusammen niemals mehr als $7n \in O(n)$ Speicher benötigen. Die drei Algorithmen haben insgesamt einen Speicherbedarf von $\Theta(n) + \Theta(nm) + O(n) = \Theta(nm)$.

Damit wächst auch der Speicherbedarf asymptotisch linear mit der Anzahl der Stellen und Anzahl der Umlegungen. Wie bei der dynamischen Programmierung üblich, wird hier ein etwas höherer Speicherbedarf für eine deutlich verkürze Laufzeit in Kauf genommen.

1.5 Beweis der Richtigkeit

1.5.1 Algorithmus 1

Zunächst wird die Richtigkeit von Algorithmus 1 per Induktion bewiesen. Zur Erinnerung: $dp[p_0][e]$ soll die minimale Anzahl an Umlegungen sein, die benötigt werden, um aus dem Teilwort $w_{p_0}, ..., w_n$ und e zusätzlichen Stäbchen eine valide Zahl zu erhalten. In Gleichung 2 sind die korrekten Werte von $f(p_0, e)$ für $p_0 = n + 1$ bereits definiert, was den Induktionsanfang bildet.

Für den Induktionsschritt wird angenommen, dass dp für Werte $p > p_0$ bereits richtig ist und bewiesen, dass auch die Werte für p_0 richtig berechnet werden. Angenommen, der Algorithmus funktioniert nicht und es gäbe eine Möglichkeit, eine valide Zahl mit weniger Umlegungen als durch Gleichung 2 gegeben zu erstellen. Jede mögliche valide Zahl muss an der Stelle p_0 eine Ziffer zwischen 0 und F haben. Jedes valide Wort benötigt also mindestens so viele Umlegungen, wie nötig werden, um die Stelle p_0 zu einer Ziffer umzulegen ($\min_{x\in[0,15]} d(w_p, x)$). Damit eine valide Zahl entsteht, muss nun das Wort $w_{p_0+1}, ..., w_n$ mit $e + n(w_p, x)$ zusätzlichen Stäbchen mit möglichst wenig Umlegungen in eine valide Hex-Zahl umgelegt werden. Daraus folgt, dass wenn $f(p_0 + 1, e + n(w_p, x))$ optimal ist, es keine Möglichkeit geben kann, die weniger als $f(p_0, e)$ Umlegungen benötigt und $f(p_0, e)$ damit optimal ist. Das Problem hat also die sogenannte *optimal substructure*. Daraus folgt per Induktion über n, dass nach Iteration i der äußeren for-Schleife $dp[p][e]$ für $n - i < p \leq n + 1$ korrekt ist. Folglich ist $dp[p][e]$ nach allen $i = n$ Iterationen für alle $1 \leq p \leq n + 1$ und damit auch Algorithmus 1 korrekt.

1.5.2 Algorithmus 2

Die Richtigkeit von Algorithmus 2 kann mit einem ähnlichen Prinzip begründet werden. Der Algorithmus soll die größte Hex-Zahl ausgeben, die mit $\leq m$ Umlegungen aus dem Wort w hergestellt werden kann. Der Beweis erfolgt über eine Invariante: Nach p Durchläufen der äußeren for-Schleife gilt (a) s ist die maximale Zahl, die mit den ersten p Stellen gebildet werden kann, (b) e ist die Anzahl der überschüssigen Stäbchen, wenn die ersten p Stellen zu s umgelegt werden und (c) z ist die Anzahl der noch übrigen Umlegungen. Mit $s = \epsilon$, $z = m$ und $e = 0$ gelten alle drei Bedingungen vor der ersten Iteration.

In jeder Iteration wird die größte Ziffer für die Stelle p zu s hinzugefügt, bei der es noch möglich ist, mit den danach noch übrigen Umlegungen das Wort $w_{p+1}, ..., w_n$ mit den dann übrigen Stäbchen zu einer validen Hex-Zahl umzulegen. Die so entstandene Zahl ist die größtmögliche Zahl, die mit den ersten p Stellen gebildet werden kann, da die ersten $p - 1$ Stellen bereits die größtmögliche Zahl waren und der Stelle p ebenfalls die größtmögliche Stelle zugewiesen wurde.

Dies kann über einen Widerspruchsbeweis gezeigt werden: Gäbe es eine größere Zahl als die gefundene, so müsste diese entweder eine größere Zahl in den ersten $p - 1$ Stellen oder an der Stelle p haben. Beides ist aber bereits maximal. Folglich kann es keine größere Zahl geben, die die ersten p Stellen verwendet. Auch dieses Problem hat *optimal substructure*.

Bedingung (a) ist also auch nach einem Durchlauf der Schleife noch korrekt. Wurde eine Ziffer für Stelle p ausgewählt, werden e und z entsprechend angepasst, sodass auch die Bedingungen (b) und (c) weiter zutreffen.

Folglich ist s nach n-Iterationen die maximale Zahl, die in den ersten n Stellen mit höchstens m Umlegungen gebildet werden kann und also ist Algorithmus 2 korrekt.

1.5.3 Algorithmus 3

Es folgt der Beweis für Algorithmus 3. Der Algorithmus erhält zwei Zahlen und soll, sofern dies möglich ist, eine Reihe von minimal vielen Umlegungen ausgeben, um von der ersten Zahl zur zweiten zu gelangen. Außerdem soll beim Durchführen dieser Umlegungen niemals eine Stelle der Zahl vollständig geleert werden. Zunächst wird bewiesen, dass der Algorithmus eine minimale Reihe von Umlegungen findet, sofern diese existiert. Eine Reihe von Umlegungen existiert genau dann, wenn beide Zahlen gleich viele Stäbchen verwenden. Sei e die Menge aller Positionen, in denen ein Stäbchen fehlt und n die Menge der Positionen, in denen ein Stäbchen liegt, aber nicht liegen sollte. Damit das Umlegungen möglich ist, muss also $|e| = |n|$ gelten. Durch die Zeilen 6-11 werden alle Positionen, in denen ein Stäbchen fehlt oder hinzugefügt werden muss während des Algorithmus e bzw. n hinzugefügt. Die while-Schleife entfernt dann immer ein Element aus beiden Mengen, sofern dies möglich ist. Da $|e| = |n|$, werden alle Elemente aus den Listen wieder entfernt und umgelegt und $|e|$ bzw. $|n|$, also die minimale Anzahl, viele Stäbchen bewegt.

Nun bleibt zu zeigen, dass bei diesen Umlegungen eine Stelle niemals vollständig geleert wird. Um eine Position zwischen zwei Umlegungen vollständig zu leeren, müssten zuerst alle Stäbchen der Stelle in andere Stellen verschoben werden, sodass die Stelle leer werden kann, und dann die nun benötigten

Stäbchen aus anderen Stellen in diese Stelle bewegt werden. Wenn eine Stelle komplett geleert wird, kann es also keine Umlegung innerhalb dieser Stelle geben. Es ist immer möglich ein Stäbchen innerhalb der Ziffer umzulegen, um zu verhindern, dass die Stelle vollständig geleert wird. Eine Ausnahme ist, wenn in einer Stelle nur Stäbchen entfernt bzw. hinzugefügt werden müssen. In diesem Fall kann die Stelle aber auch nicht geleert werden, da keine Hex-Ziffer leer ist. In den Zeilen 5-12 werden alle fehlenden bzw. überschüssigen Stäbchen den Listen e und n an deren Ende hinzugefügt. Da der Algorithmus in den Zeilen 13-17 die fehlenden bzw. überschüssigen Stellen vom Ende der Liste verwendet, wird immer eine Umlegung innerhalb der Stelle durchgeführt, sofern dies möglich ist. Folglich wird beim Durchführen der durch den Algorithmus ausgegeben Umlegungen nie ein Zeichen vollständig geleert und auch Algorithmus 3 ist korrekt.

1.6 Erweiterungen

1.6.1 Kurze Wege beim Umlegen

Nachdem Severin es geschafft hat, die größtmögliche Hex-Zahl zu finden, möchte die Lehrerin, dass er die Umlegungen jetzt auch auf dem Tisch ausführt. Da die Zahl aber sehr lang ist, muss er sich dabei viel hin und her bewegen. Daher sucht Severin nun eine Möglichkeit, die Umlegungen durchzuführen und sich dabei so wenig wie möglich zu bewegen. Er kann immer nur ein Stäbchen gleichzeitig in der Hand halten. Gesucht ist also eine Folge von abwechselnd Positionen, an denen ein Stab übrig ist und Positionen, an denen ein Stäbchen benötigt wird, bei der die Abstände zwischen benachbarten Positionen möglichst gering sind und alle Positionen besucht werden. Jedem Stäbchen wird dabei die Position des Mittelpunktes mit x- und y-Koordinate zugewiesen.

Zum Lösen dieses Problems, welches ein Spezialfall des Problems des Handlungsreisenden (TSP) ist, wird ein evolutionärer Algorithmus verwendet. Dieser findet nicht unbedingt die beste Lösung, da es sich um ein heuristisches Verfahren handelt, dafür kann der Algorithmus aber auch bei sehr großen Instanzen in angemessener Zeit eine akzeptable Lösung finden.

Evolutionäre Algorithmen sind Metaheuristiken zum Lösen von Optimierungsproblemen, die vom Prozess der Evolution in der Natur inspiriert sind. Mögliche Lösungen werden als Individuen einer Population dargestellt, die zu Beginn aus zufälligen Lösungen besteht. Danach werden mehrere Generationen simuliert. In jeder Generation wird die Fitness (also Qualität) aller Individuen der Population bestimmt. Aus den besten Individuen der letzten Generation werden dann bei der Selektion die Individuen der nächsten Generation erstellt. Dabei werden die Lösungen noch leicht zufällig verändert (Mutation), um sich schrittweise einer besseren Lösung annähern zu können. Optional kann auch aus zwei Eltern-Individuen durch Crossover ein Individuum der nächsten Generation erstellt werden. Nach einer bestimmten Anzahl an Generationen wird die beste bisher gefundene Lösung zurückgegeben.

Für dieses Problem werden mögliche Lösungen als Liste von Positionen modelliert, die Severin in dieser Reihenfolge abgehen muss. Bei jeder Position mit geradem Index wird ein Stäbchen aufgenommen und dann bei der nächsten Position (mit ungeradem Index) abgelegt. Bei der Mutation werden dann (potenziell mehrfach) zwei gerade bzw. zwei ungerade Positionen zufällig miteinander ausgetauscht. Für jedes Individuum wird die Länge des Weges berechnet, indem die Summe der euklidischen Distanzen zwischen den in der Lösung benachbarten Positionen berechnet wird. Schließlich werden alle Individuen einer Generation sortiert und die besten ausgewählt. Die Selektion findet nach Rang statt, da sich so (gegenüber z. B. der Fitness-proportionalen Selektion) auch kleine Unterschiede in den Längen durchsetzen können. Bei der Mutation muss zusätzlich noch die Bedingung beachtet werden, dass eine Stelle nie vollständig geleert werden kann. Ist dies der Fall, wird das Individuum verworfen und eine neue Mutation durchgeführt. Zusätzlich wird der sogenannte Elitismus (eng. elitism) verwendet, um zu verhindern, dass sich die Lösungen über Generationen verschlechtern. Das heißt, dass die besten n Individuen aus der letzten Generation ohne Mutation in die nächste übernommen werden. Außerdem werden in jeder Generation eine geringe Menge von vollständig zufällig generierten Individuen hinzugefügt, was dem Algorithmus helfen kann, aus einem lokalen Minimum zu entkommen. Die Population enthält zu Beginn bereits die Lösung von Algorithmus 3. So kann der evolutionäre Algorithmus diese Lösung als Ausgangspunkt verwenden und verbessern. In der Praxis findet dieser evolutionäre Algorithmus in allen Beispielen von der BWINF-Website etwas kürzere Umlegungen als Algorithmus 3.

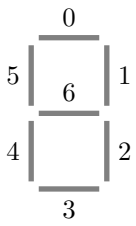


Abbildung 2: Nummerierung der Segmente. In Bit 0 wird z. B. gespeichert, ob das obere Segment ein Stäbchen hat.

2 Umsetzung

Die Lösung wurde in `C++` implementiert. Die drei Algorithmen sind in der Klasse `DPSolver` implementiert. Die Datei `main.cpp` liest die Argumente und Problemdefinition ein, ruft den `DPSolver` auf und gibt das Ergebnis aus.

Eine mögliche Belegung wird durch die Klasse `Zustand` modelliert. Jede einzelne Stelle wird kompakt als `std::bitset<7>` dargestellt. Abbildung 2 zeigt, welche Bits zu welcher Position gehören.

Die `Zustand`-Klasse enthält außerdem Funktionen zur Ein- und Ausgabe von Zuständen. Beim Einlesen müssen Hex-Ziffern in Belegungen der sieben Segmente umgewandelt werden. Dazu wird die Tabelle `STAEBCHEN` verwendet, die für jede Ziffer die Belegung, als Hex-Wert encodiert, enthält. Durch Suchen der Belegungen in `STAEBCHEN` kann ein Zustand zurück in eine Hex-Zahl umgewandelt werden. Für die Ausgabe der Zwischenzustände ist es allerdings nötig, auch Zustände auszugeben, die sich aktuell nicht in eine Hex-Zahl umwandeln lassen. Um diese dennoch in der Befehlszeile ausgeben zu können, werden die Rahmenzeichen des Unicode-Zeichensatzes verwendet. Diese Methode der Darstellung ist portabel und vergleichsweise platzsparend. Die Methoden `visualisiereZiffer()` und `visualisiereZahl()` stellen eine Ziffer bzw. eine ganze Zahl über Rahmenzeichen dar. Dabei ist jede Ziffer drei Zeilen hoch. Die Funktion `visualisiereZiffer()` gibt die drei Zeilen der Ziffer als einzelne Strings zurück, sodass mehrere Ziffern aneinandergefügt werden können. Die Methode `visualisiereUmlegungen()` visualisiert dann alle Zwischenzustände, die entstehen, wenn man die gegebenen Umlegungen nacheinander durchführt. Zusätzlich werden Linien hinzugefügt, die zeigen, von welcher Stelle zu welcher Stelle das Stäbchen bewegt wurde.

Nun zur Implementation des eigentlichen Lösungsverfahrens. Die `solve()`-Funktion der Klasse `DPSolver` initialisiert die Variablen und ruft dann die Methoden `dpVorberechen()` und `rekonstruiere()` auf, die Algorithmus 1 bzw. Algorithmus 2 implementieren. Als Memoisation-Tabelle wird ein Vektor verwendet, die Funktion `memoKey(int pos, int d)` gibt an, an welcher Position im Vektor ein Wert gespeichert ist. Zusätzlich wird die Funktion `getMemo()` verwendet, um auf die Werte im `memo`-Vektor zuzugreifen. So kann gewährleistet werden, dass für Werte, die zu viele Umlegungen verwenden ∞ zurückgegeben wird, ohne diese Werte extra zu speichern. Die Funktion `unterschied(x1, x2)` gibt sowohl $d(x_1, x_2)$ als auch $n(x_1, x_2)$ in einem `std::pair<int, int>` zurück. Ansonsten folgt die Implementation der beiden Funktionen dem oben beschriebenen Pseudocode. Algorithmus 3 ist in `getMoves()` implementiert. Die Umlegungen werden als Vektor von Paaren von Positionen zurückgegeben. Eine Position ist dabei ebenfalls ein Paar, der erste Wert gibt die Stelle und der zweite Wert die Position innerhalb der Stelle nach Abbildung 2 an.

Zum Compilieren des Codes wird ein `C++-17`-fähiger Compiler benötigt. Als Build-System wird CMake verwendet, der Code kann wie üblich über

```
cmake -S . -B build -D CMAKE_BUILD_TYPE=Release
cmake --build build
```

kompiliert und dann mit `./hexmax [-v <Algorithmus>] <Dateipfad>` ausgeführt werden. Der Pfad sollte auf die zu lösende Eingabedatei zeigen. Mit dem optionalen Parameter `-v <Algorithmus>` kann angegeben werden, dass die genauen Umlegungen berechnet werden sollen und mit welchem Algorithmus. Möglich sind `greedy` für Algorithmus 3 und `kurz` für den evolutionären Algorithmus. Der Code wurde unter Ubuntu 20.04. getestet.

2.1 Erweiterung: Kurze Wege beim Umlegen

Der Algorithmus zum Finden von Umlegungen, bei denen sich möglichst wenig bewegt werden muss, ist in der Klasse `TSPSolver` implementiert. In der `solve()`-Funktion findet sich der eigentliche Algorithmus. Ein Individuum ist, wie bei der Lösungsidee beschreiben, eine Liste von Positionen, bei denen abwechselnd ein Stäbchen aufgenommen bzw. abgelegt wird. Eine Position ist, wie beim Greedy-Algorithmus, ein `std::pair<int, int>`, das sowohl die Stelle als auch das Segment innerhalb der Stelle angibt. In der `length()`-Funktion wird die Länge des Weges für eine mögliche Lösung berechnet. Dafür werden die Strecken zwischen den Positionen addiert. Um die Strecken genau berechnen zu können, ist es wichtig, die Positionen der Stäbchen in einem Koordinatensystem festzulegen. Dafür wird jeweils der Mittelpunkt der Segmente verwendet und die Zeichen sind eine Längeneinheit breit und zwei Längeneinheiten hoch. Im `POSITIONS`-Vektor sind die x- und y-Koordinaten der Positionen festgelegt. Die eigentliche Distanz zwischen zwei Positionen wird dann über den Satz des Pythagoras ermittelt.

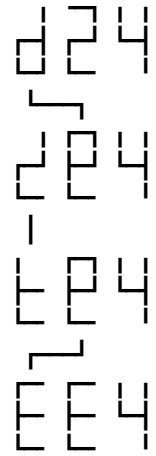
Bei evolutionären Algorithmen spielt der Zufall eine zentrale Rolle. Um effizient Pseudozufallszahlen generieren zu können wird der `std::mt19937`-Zufallszahlengenerator aus dem `random`-Header der C++-Standardbibliothek verwendet. Für die Selektion nach Rang wird eine geometrische Verteilung aus der Standardbibliothek verwendet, da diese bessere Individuen wie gewünscht bevorzugt.

Der Algorithmus ist von einigen Parametern abhängig, die im `TSPSolver`-Objekt eingestellt werden können. Standardmäßig wird eine Mutationsrate von 1% und eine Populationsgröße von 100 Individuen verwendet. Außerdem werden für jede Generation 10 Individuen der Vorgängergeneration übernommen und 10 neue Individuen generiert. Auch diese Parameter können angepasst werden.

3 Beispiele

hexmax0.txt

```
$ ./hexmax -v kurz ../Eingabe/hexmax0.txt
Die größtmögliche Hex-Zahl ist
  EE4.
Umlegungen, um zu dieser Hex-Zahl zu gelangen:
```



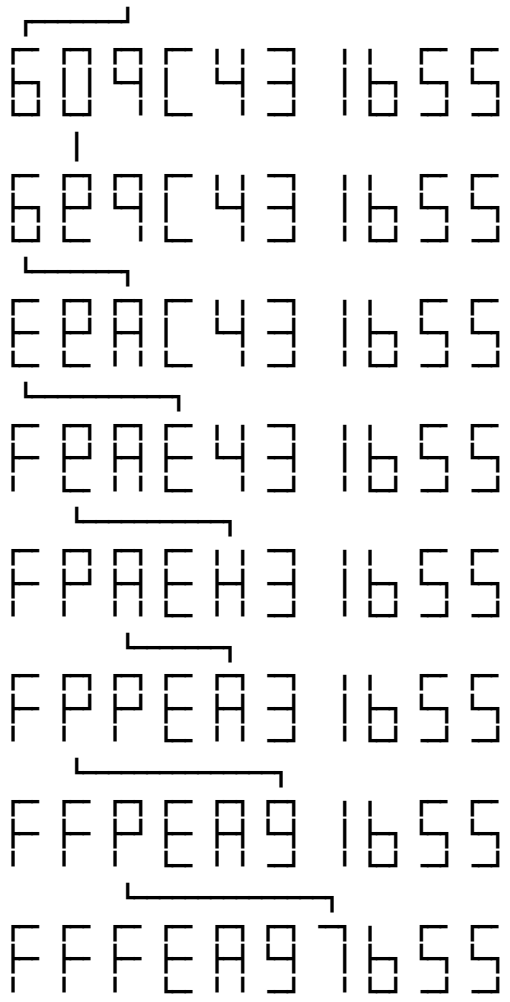
```
So muss sich Severin beim Umlegen 4.81721 Einheiten bewegen.
Mit dem Greedy-Algorithmus müsste er sich 6.82843 Einheiten bewegen.
```

Ausgaben/hexmax0_umlegungen_kurz.txt

Um Platz zu sparen sind hier nur die Umlegungen abgedruckt, die mittels evolutionärem Algorithmus einen möglichst kurzen Weg verwenden. Die Umlegungen des Greedy-Algorithmus sind in der Zip-Datei zu finden.

hexmax1.txt

```
$ ./hexmax -v kurz ../Eingabe/hexmax1.txt
Die größtmögliche Hex-Zahl ist
  FFFEA97B55.
Umlegungen, um zu dieser Hex-Zahl zu gelangen:
```



```
So muss sich Severin beim Umlegen 60.9085 Einheiten bewegen.
Mit dem Greedy-Algorithmus müsste er sich 63.4842 Einheiten bewegen.
```

Ausgaben/hexmax1_umlegungen_kurz.txt

hexmax2.txt

```
$ ./hexmax -v kurz ../Eingabe/hexmax2.txt
Die größtmögliche Hex-Zahl ist
  FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8.
Umlegungen, um zu dieser Hex-Zahl zu gelangen:
```

```
So muss sich Severin beim Umlegen 1701 Einheiten bewegen.
Mit dem Greedy-Algorithmus müsste er sich 1716.41 Einheiten bewegen.
```

Ausgaben/hexmax2_umlegungen_kurz.txt

hexmax3.txt

Für die weiteren Beispiele werden die Umlegungen – wie in den Materialen gefordert – hier nicht abgedruckt. Beide Verfahren sind jedoch in der Lage, auch für diese Beispiele Umlegungen zu generieren. Die Umlegungen sind in der Zip-Datei zu finden.

```
$ ./hexmax  ../Eingabe/hexmax3.txt
Die größtmögliche Hex-Zahl ist
  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA98BB8B9DFAFEAE888DD888AD8B
    A8EA8888.
```

Ausgaben/hexmax3.txt

hexmax4.txt

```
$ ./hexmax  ../Eingabe/hexmax4.txt
Die größtmögliche Hex-Zahl ist
  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88AD98988F898AB7AF7BDA8A61BA7D
    4AD8F888.
```

Ausgaben/hexmax4.txt

hexmax5.txt

```
$ ./hexmax  ../Eingabe/hexmax5.txt
Die größtmögliche Hex-Zahl ist
  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB899F8E8F9AA9E9AD8898
    8EDA9A99888EDAD989A8BAFD8A888888888888888888888888888888888888888888888888888888888888888888
    888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
    888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
    88888888.
```

Ausgaben/hexmax5.txt

Für `hexmax5.txt` und die weiteren Beispiele sind die Umlegungen auch nicht in der Zip-Datei zu finden, da die Ausgabe in Textform sehr groß wäre ($\approx 3\,\text{GB}$ für `hexmaxc1.txt`).

Weitere Beispiele

`hexmax5.txt` bestand aus einer 1000-stelligen Hex-Zahl und $m = 1369$. Der Algorithmus ist in der Lage, auch deutlich größere Beispiele in angemessener Zeit zu lösen. Das mit dem `beispielgenerator.py`-Skript generierte Beispiele `hexmaxc0.txt` hat $n = 5000$ und $m = 5047$. Das Beispiel `hexmaxc1.txt` hat sogar $n = 10000$ und $m = 10000$. Auch dieses Beispiel kann in unter einer Minute gelöst werden. Die Ausgaben werden bei diesen Beispielen jedoch sehr lang, weshalb sie hier nicht abgedruckt sind. Sie sind aber, zusammen mit den Eingabedateien, in der Zip-Datei zu finden.

Laufzeiten

Die folgende Tabelle zeigt die Laufzeiten, jeweils einmal ohne Ausgabe der Umlegungen, einmal mit Ausgabe von Umlegungen mit dem Greedy-Algorithmus und einmal mit Ausgabe von Umlegungen mit dem evolutionären Algorithmus.

Beispiel	ohne Umlegungen (ms)	Greedy-Umlegungen (ms)	kurze Umlegungen (ms)
`hexmax0.txt`	1.8	1.8	63.0
`hexmax1.txt`	1.9	2.0	95.5
`hexmax2.txt`	2.1	2.8	185.1
`hexmax3.txt`	4.2	9.2	503.5
`hexmax4.txt`	3.7	9.6	363.0
`hexmax5.txt`	268.4	853.2	6198.0
`hexmaxc0.txt`	4253.0	–	–
`hexmaxc1.txt`	16 900.0	–	–

Alle Beispiele von der BWINF-Website werden in wenigen Millisekunden gelöst. Beispiel 5 ist deutlich größer, wird aber auch in unter einer Sekunde gelöst. Die Laufzeit des evolutionären Algorithmus ist stark von der verwendeten Anzahl an Generationen abhängig. Je länger der Algorithmus läuft, desto besser werden die Ergebnisse.

4 Quellcode

```
#ifndef INC_3_HEXMAX_DPSOLVER_H
#define INC_3_HEXMAX_DPSOLVER_H

#include <string>
#include <bitset>
#include <unordered_map>
#include <unordered_set>

#include "Zustand.h"

class DPSolver {
private:
    int max_umlegungen;

    std::vector<int> memo;
    std::vector<int> parent;

    std::string word;
    std::vector<std::bitset<7>> word_digits;

    static std::pair<int, int> unterschied(std::bitset<7> c1, std::bitset<7> c2);

    [[nodiscard]] int memoKey(int pos, int d) const;

    std::string rekonstruiere();

    int getMemo(int pos, int surplus);

    void dpVorberechnen();
```

```

public:
    std::string solve(const std::string &ursprung, int i);

    static std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>>
    getMoves(const std::string &alt, const std::string &ziel);

};

#endif //INC_3_HEXMAX_DPSOLVER_H
```

DPSolver.h

```
#include "DPSolver.h"
#include "Zustand.h"

#include <utility>
#include <string>
#include <sstream>
#include <iostream>
#include <limits>

using namespace std;

int INFINITY = numeric_limits<int>::max() / 2;

/*
 * Findet den die größte Hex-Zahl, die mit i Umlegungen aus 'ursprung'
 * gebildet werden kann.
 */
std::string DPSolver::solve(const std::string &ursprung, int i) {
    word = ursprung;
    max_umlegungen = i;

    word_digits.reserve(word.size());
    for (char c: word) {
        word_digits.push_back(Zustand::digitFromChar(c));
    }

    memo = vector<int>((ursprung.size() + 1) * (1 + 2 * max_umlegungen), -1);
    dpVorberechnen();
    auto r = rekonstruiere();
    return r;
}

const string HEX_DIGITS = "0123456789ABCDEF";

/*
 * Rekonstruiert die größte Hex-Zahl. Vorher muss 'dpVorberechnen()'
 * aufgerufen worden sein.
 */
std::string DPSolver::rekonstruiere() {
    int ueberschuss = 0;
    int umlegungen = max_umlegungen;
    string s;
    for (int pos = 0; pos < word.size(); pos++) {
        for (int i = 15; i >= 0; i--) {
            auto r = unterschied(Zustand::digitFromChar(word[pos]), Zustand::digitFromChar(HE
    X_DIGITS[i]));
            if (getMemo(pos + 1, ueberschuss + r.second) + r.first <= umlegungen) {
                ueberschuss = ueberschuss + r.second;
                s += HEX_DIGITS[i];
                umlegungen -= r.first;
                break;
            }
        }
    }
    return s;
}

/*
 * Berechnet 'memo' vor. memo[e][p] gibt an, wie viele Umlegungen mindestens benötigt
```

```
59  * werden, um aus den letzten 'e' Stellen mit 'p' zusätzlichen Stäbchen eine valide
    * Hex-Zahl zu legen. 'p' kann dabei auch negativ sein. Das Legen von zusätzlichen
61  * Stäbchen zählt nicht als Umlegung, nur das Nehmen.
    */
63  void DPSolver::dpVorberechnen() {
        for (int pos = word.length(); pos >= 0; pos--) {
65          for (int surplus = -max_umlegungen; surplus <= max_umlegungen; surplus++) {
                if (pos == word.length()) {
67                  if (surplus == 0)
                        memo[memoKey(pos, surplus)] = 0;
69                  else
                        memo[memoKey(pos, surplus)] = INFINITY;
71                  continue;
                }
73
                int min_v = INFINITY;
75              for (int i = 15; i >= 0; i--) {
                    auto r = unterschied(word_digits[pos], Zustand::digitFromNumber(i));
77
                    auto needed_moves = getMemo(pos + 1, surplus + r.second) + r.first;
79                  if (needed_moves < min_v) {
                        min_v = needed_moves;
81                  }
                }
83
                memo[memoKey(pos, surplus)] = min_v;
85          }
        }
87  }

89  /*
    * Berechnet, (0) wie viele Umlegungen benötigt werden, um von c1 zu c2 zu
91  *                kommen. Dabei zählt das Wegnehmen von Stäbchen als Umlegung,
    *                das hinzufügen aber nicht. So werden Umlegungen zwischen
93  *                Stellen nur einmal gezählt.
    *            (1) wie viele Stäbchen dabei überschüssig werden
95  */
    pair<int, int> DPSolver::unterschied(std::bitset<7> c1, std::bitset<7> c2) {
97      size_t excess = (c1 & ~c2).count();
        size_t needed = (~c1 & c2).count();
99      auto diff = excess - needed;
        return {min(excess, needed) + max<int>(diff, 0), diff};
101 }

103 /*
    * Gibt den Wert aus 'memo' aus. Liegt der Wert außerhalb des relevanten
105 * Bereichs, wird INFINITY zurückgegeben.
    */
107 int DPSolver::getMemo(int pos, int d) {
        // Benutzt bereits zu viele Umlegungen. Der Wert muss nicht in 'memo'
109     // gespeichert werden.
        if (abs(d) > max_umlegungen)
111         return INFINITY;

113     return memo[memoKey(pos, d)];
    }
115
    int DPSolver::memoKey(int pos, int d) const {
117     return (pos * (2 * max_umlegungen + 1)) + d + max_umlegungen;
    }
119

121 /*
    * Berechnet Umlegungen, um von 'alt' auf 'ziel' zu kommen und dabei nie
123 * eine Stelle vollständig zu leeren in O(n)
    */
125 vector<pair<pair<int, int>, pair<int, int>>> DPSolver::getMoves(const std::string &alt, const
        std::string &ziel) {
        vector<pair<int, int>> excess;
127     vector<pair<int, int>> needed;

129     auto alt_state = Zustand::fromString(alt);
        auto ziel_state = Zustand::fromString(ziel);
```

```

    if (alt.length() != ziel.length()) {
        cout << "Umlegung nicht möglich, unterschiedliche Zahl an Stellen.\n";
        return {};
    }

    vector<pair<pair<int, int>, pair<int, int>>> r;
    for (int i = 0; i < alt.length(); i++) {
        for (int j = 0; j < 7; j++) {
            if (ziel_state.stellen[i][j] && !alt_state.stellen[i][j]) {
                needed.emplace_back(i, j);
            } else if (!ziel_state.stellen[i][j] && alt_state.stellen[i][j]) {
                excess.emplace_back(i, j);
            }
        }
        while (!(excess.empty() || needed.empty())) {
            r.emplace_back(excess.back(), needed.back());
            excess.pop_back();
            needed.pop_back();
        }
    }
    if (!excess.empty() || !needed.empty()) {
        cout << "Umlegung nicht möglich, unterschiedliche Anzahl an Stäbchen.\n";
        return {};
    }
    return r;
}
```

DPSolver.cpp

```
#ifndef HEXMAX_TSPSOLVER_H
#define HEXMAX_TSPSOLVER_H

#include <string>
#include <vector>
#include <random>
#include "Zustand.h"

class TSPSolver {
    typedef std::vector<std::pair<int, int>> Individuum;
    std::mt19937 rng;

private:
    void shuffle(Individuum &individuum);

    Individuum mutate(const Individuum &individuum);

    static float length(Individuum individuum);

    static bool is_valid(Individuum ind, const Zustand &state);

public:
    TSPSolver();

    // Anzahl der Mutationen
    float MUTATION_RATE = 0.01;
    // Größte der Population
    int POPULATION_SIZE = 100;
    // Anzahl der Individuen, die ohne Veränderung übernommen werden
    int ELITES = 10;
    // Anzahl der vollständig zufälligen Individuen in jeder Generation
    int NEW_INDIVIDUUMS = 10;

    std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>>
    solve(const std::string &from, const std::string &to);

    static float length(const std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>>
     &moves);
};

#endif //HEXMAX_TSPSOLVER_H
```

TSPSolver.h

```cpp
#include <iostream>
#include <algorithm>
#include <iomanip>
#include "TSPSolver.h"
#include "Zustand.h"
#include "DPSolver.h"

using namespace std;


/*
 * Findet Umlegungen, bei denen sich möglichst wenig bewegt werden muss über
 * einen evolutionären Algorithmus
 */
std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>>
TSPSolver::solve(const std::string &from, const std::string &to) {

    vector<Individuum> population;

    auto from_state = Zustand::fromString(from);

    // Umlegungen des Greedy-Algorithmus als Ausgangspunkt verwenden
    auto moves_greedy = DPSolver::getMoves(from, to);
    Individuum individuum;
    for (auto &move: moves_greedy) {
        individuum.push_back(move.first);
        individuum.push_back(move.second);
    }
    population.push_back(individuum);

    // Den Rest mit zufälligen Individuuen initialisieren
    while (population.size() < POPULATION_SIZE) {
        Individuum u = individuum;
        shuffle(u);
        if (is_valid(u, from_state))
            population.push_back(u);
    }

    Individuum best = individuum;

    for (int generation = 0; generation < 1000; generation++) {

        // Längen der Wege berechnen und sortieren
        vector<pair<int, Individuum>> lengths;
        for (const auto &ind: population) {
            lengths.emplace_back(length(ind), ind);
        }
        std::sort(lengths.begin(), lengths.end());
        best = lengths[0].second;

        // Verteilung zur Selektion
        geometric_distribution dist(0.05);
        population.clear();


        int avg_dist = 0;
        for (int i = 0; i < ELITES; i++) {
            population.push_back(lengths[i].second);
            avg_dist += lengths[i].first;
        }

        for (int i = 0; i < NEW_INDIVIDUUMS; i++) {
            Individuum u = individuum;
            shuffle(u);
            if (is_valid(u, from_state))
                population.push_back(u);
        }

        while (population.size() < POPULATION_SIZE) {
            auto u = mutate(lengths[min<int>(dist(rng), lengths.size() - 1)].second);
            if (is_valid(u, from_state))
                population.push_back(u);
        }
```

```
74      }
76      // Rückgabewert aus 'best' erstellen
        vector<pair<pair<int, int>, pair<int, int>>> moves;
78      for (int i = 0; i < best.size(); i += 2) {
            moves.emplace_back(best[i], best[i + 1]);
80      }
        return moves;
82  }
84  TSPSolver::TSPSolver() {
        random_device dev;
86      rng = mt19937(dev());
    }
88
    /*
90   * Mischt eine Lösung, um eine vollständig zufällige Lösung zu
     * generieren. Dabei dürfen nur gerade mit geraden und ungerade mit
92   * ungeraden Positionen getauscht werden, da bei geraden Positionen
     * ein Stäbchen aufgenommen und bei ungeraden ein Stäbchen abgelegt
94   * wird.
     */
96  void TSPSolver::shuffle(TSPSolver::Individuum &individuum) {
        vector<pair<int, int>> needed, excess;
98      for (int i = 0; i < individuum.size(); i += 2) {
            excess.push_back(individuum[i]);
100         needed.push_back(individuum[i + 1]);
        }
102     std::shuffle(needed.begin(), needed.end(), rng);
        std::shuffle(excess.begin(), excess.end(), rng);
104     for (size_t i = 0; i < needed.size(); i++) {
            individuum[2 * i] = excess[i];
106         individuum[2 * i + 1] = needed[i];
        }
108 }
110 /*
     * Mutiert das Individuum durch Tauschen von Elementen
112  */
    TSPSolver::Individuum TSPSolver::mutate(const TSPSolver::Individuum &individuum) {
114     Individuum child = individuum;
        uniform_int_distribution<int> dist(0, (individuum.size() / 2) - 1);
116     uniform_int_distribution<int> bool_dist(0, 1);
        for (int i = 0; i < max(1, (int) (MUTATION_RATE * individuum.size())); i++) {
118         int type = bool_dist(rng);
            swap(child[dist(rng) * 2 + type], child[dist(rng) * 2 + type]);
120     }
        return child;
122 }
124 /*
     * Überprüft, ob bei den Umlegungen eine Stelle vollständig geleert wird
126  */
    bool TSPSolver::is_valid(TSPSolver::Individuum ind, const Zustand &old_state) {
128     vector<int> counts;
        for (const auto &pos: old_state.stellen) {
130         counts.push_back(pos.count());
        }
132     for (int i = 0; i < ind.size(); i += 2) {
            counts[ind[i].first] -= 1;
134         counts[ind[i + 1].first] += 1;
            if (counts[ind[i].first] <= 0) {
136             return false;
            }
138     }
        return true;
140 }
142 /*
     * Berechnet die Länge des Weges beim Umlegen
144  */
    float TSPSolver::length(TSPSolver::Individuum individuum) {
146     float len = 0;
```

```
    pair<int, int> current_pos = individuum[0];
    // Positionen der Stäbchen innerhalb einer Stelle
    const vector<array<float, 2>> POSITIONS = {
            {0.5, 0},
            {1,   0.5},
            {1,   1.5},
            {0.5, 2},
            {0,   1.5},
            {0,   0.5},
            {0.5, 1}
    };
    for (int i = 1; i < individuum.size(); i++) {
        // Distanz über Satz des Pythagoras berechnen
        auto start_pos = POSITIONS[current_pos.second];
        start_pos[0] += current_pos.first;
        auto end_pos = POSITIONS[individuum[i].second];
        start_pos[0] += individuum[i].first;
        len += sqrt(pow(end_pos[0] - start_pos[0], 2) + pow(end_pos[1] - start_pos[1], 2));
        current_pos = individuum[i];
    }
    return len;
}

float TSPSolver::length(const std::vector<std::pair<std::pair<int, int>, std::pair<int, int>>>
     &moves) {
    Individuum ind;
    for (const auto move: moves) {
        ind.push_back(move.first);
        ind.push_back(move.second);
    }
    return length(ind);
}
```

TSPSolver.cpp