Seilschaften

Aufgabe 1, Runde 2

Marian Dietz

1 Lösungsidee

1.1 Begriffserklärungen

Personen und Steine, also „Objekte“, die in den zwei Körben sein und hoch- bzw. runterfahren können, werden unter dem Begriff *Komponente* zusammengefasst.

Die *Gewichtsschranke*, also die Maximaldifferenz zwischen den Gewichten der beiden Körbe, ist d.

Eine *Seite* einer Komponente kann entweder *oben* oder *unten* sein. Dabei wird nicht berücksichtigt, ob sich die Komponente im Korb befindet, oder nicht.

Der Ort, an dem sich eine Komponente befindet bzw. befinden kann, wird als *Position* bezeichnet. Der Unterschied zu einer Seite ist, dass eine Position zusätzlich zwischen „im Korb“ und „nicht im Korb“ unterscheidet. Außerdem kann eine Position nicht genau angegeben (egal) sein. Der obere Index einer Position zeigt an, ob sie oben oder unten ist, der untere Index zeigt an, ob sie im Korb oder außerhalb des Korbes (frei) ist. Die Liste zeigt die möglichen Positionen.

- p_f^o bzw. p_f^u - die Position muss oben bzw. unten außerhalb des Korbes sein.
- p_k^o bzw. p_k^u - die Position muss oben bzw. unten im Korb sein.
- p_e^o bzw. p_e^u - die Position muss oben bzw. unten sein, dabei ist jedoch egal, ob sie im Korb ist, oder nicht.
- p_f^e bzw. p_k^e - die Position muss außerhalb des Korbes bzw. im Korb sein, dabei ist egal, auf welcher Seite sie ist.
- p_e^e - es ist egal, ob die Position oben oder unten ist, und ob sie im Korb ist, oder nicht.

Eine *spezifische Position* ist eine Position, bei der genau festgelegt ist, auf welcher Seite die Position ist, und ob sie im Korb ist, oder nicht. Mögliche spezifische Positionen sind also nur p_f^o, p_f^u, p_k^o und p_k^u.

Eine *Startposition* ist die Position einer Komponente, die sie zu Beginn hat.

Eine *Endposition* ist die Position einer Komponente, die sie am Ende haben soll.

Ein *Status* beinhaltet Positionen aller Komponenten.

Bei einer *Fahrt* fährt ein Korb nach unten, der andere nach oben. Damit dies möglich ist, müssen Komponenten in den Körben sein.

Bei einem *Übergang* von einem Status zu einem anderen ändern sich die Positionen der Komponenten vom ersten Status zu denen des anderen, ohne, dass eine Fahrt durchgeführt werden muss. Ein solcher Übergang ist logischerweise nur möglich, wenn keine Komponente die Seite wechselt. Außerdem können Steine sich nicht alleine in einen Korb rein- bzw. aus einem Korb rausbewegen. Dafür muss eine Person auf der entsprechenden Seite anwesend sein.

Eine *kürzeste Lösung* ist eine Folge von Fahrten, die zum in der Aufgabe genannten Ergebnis führt, wobei es nicht möglich ist, eine andere solche Lösung zu finden, die eine kleinere Anzahl von Fahrten hat.

1.2 Erweiterungen

Die Aufgabenstellung wurde so erweitert, dass eine Startposition jede beliebige spezifische Position sein kann. Die eigentliche Aufgabenstellung gibt nur vor, dass jede Komponente am Anfang entweder *oben* oder *unten* sein kann. Dabei wird auch nicht genau gesagt, ob diese Positionen im Korb oder außerhalb des Korbes sind. Ich gehe jedoch davon aus, dass damit gemeint ist, dass alle Komponenten zu Beginn außerhalb des Korbes sind. Es kommen also noch p_k^o und p_k^u zu den möglichen Startpositionen hinzu.

Dass nur spezifische Positionen als Startpositionen verwendet werden können, liegt daran, dass Komponenten am Anfang eine *genaue* Position benötigen. Es reicht z. B. nicht aus, wenn man nur weiß, dass eine Komponente oben ist, denn dann ist unklar, ob sie sich im Korb befindet, oder nicht. Ein Beispiel, wobei die hinzugekommenen Startpositionen sehr sinnvoll sind ($d = 15$):

Art	**Gewicht**	**Startposition**
Person	50	p_f^o
Stein	35	p_k^u

Die Person kann nur nach unten fahren, wenn der Stein hochfährt. Dafür muss der Stein aber im Korb sein, wofür standardmäßig eine Person anwesend sein müsste. Der Stein kann dank der Erweiterung aber bereits zu Beginn im Korb sein und daher ungehindert die Person nach unten bringen.

Außerdem ist es durch die Erweiterungen möglich, Endpositionen anzugeben, was die Aufgabenstellung eigentlich nicht vorschreibt. Dafür ist sogar, anders als bei den Startpositionen, jede beliebige Position möglich, sie müssen nicht spezifisch sein. Auch diese Erweiterung ist sehr sinnvoll, da man bspw. angeben kann, dass bestimmte Personen am Ende oben sein sollen. Oder wenn es komplett egal ist, wo eine Komponente zum Schluss sein soll, wird die Position p_e^e verwendet. Wird keine Endposition für eine Komponente angegeben, werden die standardmäßigen Positionen aus der originalen Aufgabenstellung

verwendet. Steine bekommen standardmäßig wie in der Aufgabenstellung beschrieben die Endposition p_e^e, Personen hingegen p_e^u.

Die Erweiterungen, insbesondere die Endpositionen, die abgeändert werden können, erschweren die Realisierung deutlich, wie man z. B. im Abschnitt 1.5.3 („Sonderfall“) sehen kann.

1.3 Berechnung der möglichen Fahrten

Wie der Algorithmus zur Berechnung einer kürzesten Lösung funktioniert, soll an folgendem Beispiel gezeigt werden ($d = 5$):

ID	Art	Gewicht	Startposition	Endposition
0	Person	20	p_f^o	p_e^e
1	Stein	15	p_f^u	p_e^o
2	Person	80	p_f^o	p_e^u
3	Person	75	p_f^o	p_e^u

Zunächst müssen alle möglichen Fahrten berechnet werden, damit diese später immer wieder durchprobiert werden können. Das funktioniert folgendermaßen:

Von der Menge mit allen Komponenten wird die Potenzmenge gebildet. Alle Elemente der Potenzmenge werden einmal mit allen anderen Elementen der Potenzmenge verknüpft. Dabei entstehen also geordnete Paare aus zwei Mengen, in denen wiederum Komponenten enthalten sind. Die erste Menge entspricht den Komponenten, die bei einer Fahrt runterfahren, die andere Menge entspricht den Komponenten, die bei einer Fahrt hochfahren. Jetzt muss überprüft werden, ob die Fahrt auch so gültig ist. Dafür müssen folgende Bedingungen gelten, wobei w_1 das Gesamtgewicht der Komponenten, die runterfahren und w_2 das Gesamtgewicht der Komponenten ist, die hochfahren:

- Es ist keine Komponente in beiden Mengen gleichzeitig enthalten. Das würde nämlich bedeuten, dass eine Komponente gleichzeitig hoch- und runterfährt, was logischerweise nicht möglich ist.
- $w_1 - w_2 > 0$, denn der obere Korb muss schwerer belastet werden, um runterfahren zu können.
- $w_1 - w_2 \leq d$ **oder** es sind in beiden Mengen nur Steine enthalten, d. h., nur Steine nehmen an der Fahrt teil. Steine dürfen nämlich ohne eine Gewichtsschranke fahren. Wenn jedoch mindestens eine Person dabei ist, muss die Differenz zwischen den Gewichten der Körbe kleiner oder gleich der Gewichtsschranke d sein.

Werden alle Bedingungen eingehalten, so ist die Fahrt möglich.

Bei dem oben genannten Beispiel würden folgende Fahrten durch die zugehörigen geordneten Paare $(\{0\},\{1\})$, $(\{2\},\{3\})$ und $(\{3\},\{\})$ entstehen, wobei die Zahlen die IDs der zugehörigen Komponenten sind:

nach unten	nach oben
0	1
2	3
3	-

Wurden auf diese Weise alle möglichen Fahrten berechnet, kommt der nächste Schritt.

1.4 Breitensuche

Mithilfe einer Breitensuche wird **rückwärts** eine kürzeste Lösung gesucht. Dafür wird eine Warteschlange benutzt, in der zunächst nur der Status mit allen Endpositionen enthalten ist:

ID	Endposition
0	p_e^e
1	p_e^o
2	p_e^u
3	p_e^u

Jetzt wird immer wieder der vorderste Status der Warteschlange entnommen. Bei jedem Durchgang werden alle unter „Berechnung der möglichen Fahrten“ berechneten Fahrten rückwärts auf diesen Status ausprobiert. Bevor jedoch eine Fahrt durchgeführt werden kann, muss ein Übergang stattfinden. Dieser startet beim alten Status, der aus der Warteschlange geholt wurde, und endet bei demjenigen Status, der für die Fahrt benötigt wird. Wie genau ein Übergang und das Durchführen von Fahrten funktionert, wird unter „Übergang“ bzw. „Ausführen von Fahrten“ erklärt.

- Ist der Übergang nicht möglich, kann die Fahrt auch nicht stattfinden.
- Ist der Übergang möglich, so wird die Fahrt *rückwärts* durchgeführt, da der Algorithmus rückwärts nach einer Lösung sucht. Der Status, der durch die rückwärts durchgeführte Fahrt entsteht, wird der Warteschlange hinzugefügt, damit später, wenn der Algorithmus bei diesem Status in der Warteschlange ankommt, wieder alle möglichen Fahrten ausprobiert werden können.

Immer, bevor Fahrten mit einem Status ausprobiert werden, wird der Status, der der Warteschlange entnommen wurde, einer Liste hinzugefügt. Wenn später ein Status aus der Warteschlange geholt wird, wird zunächst überprüft, ob dieser schon in der Liste enthalten ist. Wenn ja, kann dieser übersprungen werden, da mit ihm schon einmal alle Fahrten durchprobiert wurden. Es wäre sinnlos, das ganze noch einmal zu tun.

Soll irgendwann ein Element aus der Warteschlange entnommen werden, wenn keines mehr enthalten ist, gibt es keine Lösung, da alle Fahrten durchprobiert wurden, es aber nirgends weitergehen kann bzw. alle Status, wo man weitersuchen könnte, bereits

komplett durchprobiert wurden, weil sie in der Liste mit allen schon besuchten Status enthalten sind.

Außerdem wird immer, bevor alle Fahrten mit einem Status durchprobiert werden, überprüft, ob der aktuelle Status möglicherweise das Ziel, also der eigentliche Anfang der Aufgabe ist. Das würde bedeuten, dass eine Lösung gefunden wurde. Dies geschieht, indem wieder ein Übergang überprüft wird. Er startet bei demjenigen Status, der der Warteschlange entnommen wurde, und endet mit dem Ziel. Das Ziel ist der Status mit allen Startpositionen. Ist der Übergang vom aktuellen Status zum Ziel möglich, dann wurde eine kürzeste Lösung gefunden, denn es konnte eine Folge von Fahrten herausgesucht werden, mit denen man die Aufgabe lösen kann, ohne, dass eine weitere Fahrt durchgeführt werden muss, denn dies ist, wie unter „Begriffserklärungen“ beschrieben, eine Bedingung des Übergangs. Die Lösung besteht also aus all den Fahrten, die zum aktuellen Status geführt haben, natürlich in der richtigen Reihenfolge und rückwärts. Ist der Übergang nicht möglich, so wurde noch keine Lösung gefunden und es wird einfach weitergesucht.

Es wird **immer** eine *kürzeste* Lösung gefunden. Das liegt daran, dass am Anfang nur der Status mit allen Endpositionen in der Warteschlange enthalten ist. Es werden danach nur weitere Status der Warteschlange hinzugefügt, zu denen jeweils nur eine einzige Fahrt führt, natürlich von hinten aus gesehen, da der Algorithmus rückwärts sucht. Während diese wiederum bearbeitet werden, kommen in die Warteschlange nur Status, zu denen insgesamt zwei Fahrten führen. Die Anzahl der Fahrten steigt in der Warteschlange also immer weiter und es werden bei jedem Status stets **alle** möglichen Fahrten durchprobiert. Sobald ein Status gefunden wurde, von dem ein Übergang zum Ziel möglich ist, wird die Breitensuche abgebrochen. Es kann demnach keine noch kürzere Lösung existieren.

1.5 Übergang

Ein Übergang ist nur möglich, wenn keine Komponente die Seite wechselt. Außerdem können Steine nicht ohne anwesende Person in einen Korb eingeladen bzw. aus einem Korb ausgeladen werden.

1.5.1 Erstellen eines für eine Fahrt benötigten Status

Um den Übergang zu einer Fahrt zu testen, müssen zunächst die Positionen berechnet werden, die die Komponenten nach der Fahrt überhaupt haben sollen. Die Komponenten müssen von den *aktuellen* zu den *benötigten* Positionen wechseln. Die folgende Tabelle zeigt die aktuellen und die benötigten Positionen für den Übergang vom Status mit allen Endpositionen zu dem Status, der für die erste Fahrt benötigt wird, bei der Komponente null nach unten und Komponente eins nach oben fährt. Die benötigten Positionen sind immer diejenigen Positionen, die die Komponenten direkt nach der Fahrt haben.

ID	benötigte Position	aktuelle Position
0	p_k^u	p_e^e
1	p_k^o	p_e^o
2	p_f^u	p_e^u
3	p_f^u	p_e^u

Folgendermaßen wird einer Komponente die benötigte Position zugewiesen:

1. Wenn eine Komponente bei der Fahrt nach unten fährt, dann bekommt sie die benötigte Position p_k^u, da sie direkt nach der Fahrt im unteren Korb ist. Dies ist bei der Komponente mit der ID null der Fall, denn sie fährt nach unten.

2. Wenn eine Komponente bei der Fahrt nach oben fährt, dann bekommt sie die benötigte Position p_k^o, da sie direkt nach der Fahrt im oberen Korb ist. Dies ist bei der Komponente mit der ID eins der Fall, denn sie fährt nach oben.

3. Nimmt eine Komponente an der Fahrt nicht teil, dann darf sie direkt nach der Fahrt nicht im Korb sitzen, wechselt aber natürlich auch nicht die Seite. D. h., die benötigte Position ist auf derselben Seite wie die aktuelle Position, hat aber im unteren Index f zu stehen, da die Komponente nach der Fahrt nicht im Korb sitzt. Dies ist bei den Komponenten zwei und drei der Fall. Sie bekommen demnach beide die Position p_f^u, da sie auf der unteren Seite außerhalb des Korbes sein müssen. Um noch ein weiteres Beispiel für diesen Fall zu nennen: eine Komponente mit der aktuellen Position p_e^e würde die benötigte Position p_f^e erhalten, wenn sie nicht an der Fahrt teilnimmt.

Dieser Schritt wird nur ausgeführt, wenn der Übergang zu einer Fahrt durchgeführt werden soll. Wenn geschaut wird, ob das Ziel erreicht wurde, steht nämlich schon der Status mit den benötigten Positionen bereit.

1.5.2 Überprüfung des Übergangs

Jetzt kann der Übergang für jede Komponente einzeln überprüft werden:

1. Wenn die benötigte und die aktuelle Position auf derselben Seite sind, dann ist der Übergang für diese Komponente mindestens eingeschränkt möglich. Dies trifft auch zu, wenn die aktuelle Position im oberen Index e hat, denn dann ist die aktuelle Seite der Komponente egal.
   a) Wenn außerdem die unteren Indizes der beiden Positionen übereinstimmen (beide Positionen sind im Korb bzw. außerhalb), dann stimmen die Positionen voll überein und der Übergang für diese Komponente ist auf jeden Fall möglich. Dies trifft auch zu, wenn die aktuelle Position im unteren Index e hat, denn dann ist es egal, ob die Komponente zurzeit im Korb ist, oder nicht.

   b) Tritt Fall a) nicht in Kraft, dann ist eine der beiden Positionen im Korb, die andere jedoch nicht. Daher wird auf der entsprechenden Seite eine Person benötigt, um diese Komponente in den Korb rein- bzw. aus dem Korb rauszubefördern. Auch für Personen kann man diese Bedingung aufstellen, trotz der Tatsache, dass eigentlich nur Steine Hilfe benötigen, denn Personen können sich selber in den Korb rein- bzw. aus dem Korb rausbefördern.

2. Wenn die benötigte und die aktuelle Position auf unterschiedlichen Seiten sind, ist es *nicht* möglich, den Übergang zu vollziehen, weil eine Fahrt benötigt werden würde, um die Komponente auf die entsprechende Seite zu bekommen.

Um zu überprüfen, ob auf einer Seite eine Person anwesend ist, werden einfach die Seiten (obere Indizes) der *benötigten* Positionen aller Personen angeschaut.

Wenn der Übergang für alle Komponenten möglich ist und überall, wo eine Person benötigt wird, eine Person anwesend ist, dann funktioniert der Übergang und die Fahrt kann, sofern vorhanden, rückwärts auf den Status mit den benötigten Positionen ausgeführt werden.

Im Beispiel trifft für alle Komponenten der Fall 1/a) zu, weshalb der Übergang funktioniert und nirgends eine Person benötigt wird.

1.5.3 Sonderfall

Es kann wegen der Erweiterung passieren, dass eine Person auf einer Seite benötigt wird, wo jedoch keine anwesend ist. Stattdessen gibt es aber eine Person, deren benötigte Seite egal und demnach unbekannt sind. Dies ist daran zu erkennen, dass von der Position der obere Index e ist.

Trifft das zu, dann werden einfach, direkt nachdem die Überprüfungen des Übergangs stattgefunden haben, alle Personen rausgesucht, die eine unbekannte benötigte Seite haben. Immer, wenn eine solche Person gefunden wurde, wird die benötigte Position dieser Person auf die entsprechende Seite gesetzt (natürlich außerhalb des Korbes, denn die Person nimmt nicht an der Fahrt teil). Mit der neuen Position wird ein neuer Status erstellt. Auf *jeden* dieser Status wird anschließend, sofern vorhanden, die Fahrt rückwärts durchgeführt und das Ergebnis wird anschließend der Warteschlange hinzugefügt. Es können also durch nur einen Übergang mehrere neue Status entstehen, wenn mehrere Personen unbekannte Seiten haben.

Werden sogar oben *und* unten Personen benötigt, damit Steine umgeladen werden können, müssen zwei Personen mit unbekannten Seiten existieren, damit der Übergang möglich ist. D. h., es werden alle Personen mit unbekannten Seiten untereinander kombiniert, für jede Kombination wird ein neuer Status erstellt und die benötigte Position wird für eine Person auf p_f^o, für die andere auf p_f^u gesetzt. Dabei ist wichtig, dass die beiden Personen natürlich auch einmal „tauschen“ müssen. D. h., wenn zwei Personen ausgewählt wurden, muss jede davon einmal bei einem Status unten, bei dem anderen oben sein.

Ein Beispiel für einen solchen Fall: ein Stein hat die benötigte Position p_k^u und die aktuelle Position p_f^u. Bei der Überprüfung des Übergangs trifft der Fall 1/b) zu, es wird

unten also eine Person benötigt. Es ist jedoch keine anwesend, es gibt aber zwei Personen mit der benötigten Position p_f^e. Es werden alle Personen rausgesucht, deren Seiten egal sind. Dies sind die beiden eben genannten Personen. Es werden bei diesem Übergang also zwei Status erstellt. Beim ersten ist die Position der ersten Person jetzt p_f^u, beim zweiten hat die zweite Person die neue Position p_f^u. Auf beide Status wird natürlich die Fahrt rückwärts durchgeführt und die dabei entstehenden Status der Liste hinzugefügt.

1.6 Durchführen von Fahrten

Die Fahrt wird nicht rückwärts durchgeführt, wenn der Übergang nicht funktioniert.

Die Fahrt wird einmal durchgeführt, wenn der Übergang ganz normal funktioniert, ohne dass Personen mit unbekannten Seiten rausgesucht werden mussten.

Die Fahrt kann sogar mehrmals durchgeführt werden, wenn durch das Aussuchen von Personen mit unbekannten Seiten mehrere Status entstanden sind. Die Fahrt wird dann auf jeden dieser Status ausgeführt.

Um eine Fahrt rückwärts durchzuführen, wird der Status mit den benötigten Positionen genommen, denn dorthin haben sich die Komponenten durch den Übergang bewegt. Alle Komponenten, die runterfahren, haben die Position p_k^u und bekommen jetzt die Position p_k^o, da der Algorithmus rückwärts durchgeführt wird. Alle Komponenten, die hochfahren, haben die Position p_k^o und bekommen jetzt die Position p_k^u.

1.7 Alternative Möglichkeiten

Die möglichen Fahrten werden absichtlich einmal am Anfang des Algorithmus berechnet. Da die Potenzmenge gebildet werden muss und alle Elemente davon einmal mit allen anderen Elementen kombiniert werden müssen, kann das bei vielen Komponenten sehr aufwendig sein und benötigt insgesamt $(2^n)^2$ Durchführungen, wobei n die Anzahl der Komponenten ist. Für alle vorgegebenen Beispiele reicht das jedoch völlig aus, und wer genug Zeit und Speicher mitbringt, kann so auf jeden Fall auch Probleme mit mehr als neun Komponenten lösen.

Die Alternative wäre gewesen, dass immer, wenn ein Status aus der Warteschlange genommen wird, alle Fahrten speziell dafür neu berechnet werden. Das kann viel schneller gehen, als am Anfang einmal alle möglichen Fahrten zu bilden, da nur die Teilmengen von den einzelnen Seiten mit den sich dort befindenden Komponenten berechnet werden müssen. Trotzdem bringt das über die gesamte Lösung gesehen meistens kein Vorteil, da dies immer wieder durchgeführt werden müsste.

2 Umsetzung

Das Programm wurde objektorientiert in der Programmiersprache Java geschrieben. Zum Ausführen muss der Befehl „java -jar [Pfad]“ in einem Terminal eingegeben werden. Es wird Java 7 oder neuer benötigt.

2.1 Klassen

Der Code besteht aus folgenden Klassen:

- `Position` - Enum, beinhaltet die Positionen `TOP_FREE` (für p_f^o), `BOTTOM_FREE` (für p_f^u), `TOP_BASKET` (für p_k^o), `BOTTOM_BASKET` (für p_k^u), `TOP` (für p_e^o), `BOTTOM` (für p_e^u), `FREE` (für p_f^e), `BASKET` (für p_k^e) und `UNKNOWN` (für p_e^e).
- `Component` - abstrakte Klasse, speichert nur das Gewicht, die Startposition und die Endposition für eine Komponente, welche durch Getter-Methoden abgefragt werden können.
- `Person` - erbt von `Component`, stellt eine Person dar, erweitert die Elternklasse um nichts.
- `Stone` - erbt von `Component`, stellt einen Stein dar, erweitert die Elternklasse um nichts.
- `ComponentList` - beinhaltet eine Liste mit Komponenten und zwei Listen mit den Start- und Endpositionen derer (diese werden anhand der Positionen erstellt, die in den Komponenten selber gespeichert sind, damit sie später einfach abgefragt werden können und nicht neu erstellt werden müssen).
- `Ride` - stellt eine Fahrt dar, beinhaltet Listen mit den Indizes der Komponenten aus der `ComponentList`, die hoch- bzw. runterfahren.
- `PossibleRidesGenerator` - generiert alle möglichen Fahrten anhand einer `ComponentList` und dem maximalen Gewichtsunterschied.
- `State` - stellt einen Status dar, beinhaltet eine Liste mit den Positionen aller Komponenten und eine Liste mit den Fahrten, die von hinten zu diesem Status führen.
- `Transition` - prüft den Übergang zwischen zwei Status.
- `SolutionFinder` - sucht nach einer kürzesten Lösung, benutzt dafür eine Warteschlange für eine Breitensuche.
- `Seilschaften` - Hauptklasse, liest die Eingaben ein, verwendet die anderen Klassen, um eine kürzste Lösung zu finden und gibt diese aus.

2.2 Funktionsweise

Das Programm startet mit der Hauptklasse `Seilschaften`, in der Methode `start()`. Dort werden die Methoden `requestDifference()` und `requestComponents()` zum Einlesen der Maximaldifferenz und der Komponenten mithilfe eines `Scanner`s (Package `java.util`) verwendet. Wie genau das geschieht, ist jedoch für den Algorithmus irrelevant. Jedenfalls werden beim Einlesen der Komponenten `Person`s und `Stone`s erstellt, welche einer `ComponentList` hinzugefügt werden. Die eingelesen Daten werden anschließend zur Sicherheit noch einmal in der Methode `printInputs()` ausgegebenen.

In `start()` wird weitergemacht, indem zunächst ein `PossibleRidesGenerator` erstellt wird, der durch die Methode `generate()` eine Liste mit allen möglichen Fahrten liefert. Danach wird ein `SolutionFinder` erstellt, der in der Methode `execute()` eine kürzeste Lösung raussucht. Diese wird dann, immer noch in `start()`, ausgegeben, sofern sie existiert. Damit ist das Programm dann beendet.

2.3 Erstellen der Fahrten

Das Berechnen der möglichen Fahrten geschieht im `PossibleRidesGenerator`, der mit einer `ComponentList` und dem maximalen Gewichtsunterschied als `int` erstellt wird: In `generate()` wird zunächst `powerSet()` aufgerufen, wo die Potenzmenge der Komponenten gebildet wird. Diese wird in `generatePossibleRides()` verwendet, um die möglichen Fahrten zu berechnen, die dann als eine Liste in `generate()` zurückgegeben werden.

`powerSet()` erstellt als Potenzmenge eine `HashMap`, dessen Schlüssel die jeweilige Teilmenge und der zugehörige Wert das Gesamtgewicht dieser Teilmenge ist, welches schon in dieser Methode berechnet wird, damit es dann durchgehend zur Verfügung steht, wenn die Teilmengen miteinander kombiniert werden. Die `Map` wird aufgebaut, indem ihr zunächst die leere Teilmenge hinzugefügt wird (logischerweise mit dem Gewicht null). Jetzt werden alle Komponenten durchlaufen. Bei jedem Durchgang werden wiederum alle bereits vorhandenen Teilmengen durchlaufen. Von jeder davon wird eine Kopie erstellt, welcher die aktuelle Komponente hinzugefügt wird. Diese Kopie wird mit dem in der Methode `weight()` berechneten Gewicht der Potenzmenge hinzugefügt. Das Gewicht wird berechnet, indem einfach alle Gewichte der Komponenten aus der Teilmenge zusammenaddiert werden. Wurden alle Komponenten durchlaufen, wird die `Map`, die jetzt alle Teilmengen und die zugehörigen Gewichte beinhaltet, zurückgegeben.

Die Methode `generatePossibleRides()` erstellt alle möglichen Fahrten anhand der übergebenen Potenzmenge. Dafür wird eine zunächst leere Liste mit `Ride`s erstellt, in die später die Fahrten eingefügt werden sollen. Alle Teilmengen werden mithilfe von verschachtelten Schleifen miteinander kombiniert. Für jede Kombination muss überprüft werden, ob alle Bedingungen für eine korrekte Fahrt eingehalten werden:

- Die Methode `containSameComponents()` überprüft, ob eine Komponente dabei ist, die sich in beiden Teilmengen befindet. Das wird mit verschachtelten Schleifen, die die Komponenten vergleichen und bei einem Fund `true` zurückgeben, überprüft. Ist dies der Fall, wird die aktuelle Kombination mit `continue` übersprungen und die Überprüfungen fahren mit der nächsten potentiellen Fahrt fort.

- Als nächstes wird die Differenz der Gewichte der beiden Teilmengen berechnet. Die Kombination wird übersprungen, wenn das Gewicht der Komponenten, die hochfahren, größer oder gleich dem Gewicht der Komponenten ist, die runterfahren. Die Gewichte können einfach aus dem Eintrag der Potenzmenge abgelesen werden, da sie schon in `powerSet()` berechnet wurden.

- Außerdem muss entweder die Differenz höchstens so groß wie die Maximaldifferenz sein, oder es fahren nur Steine mit. Letzteres wird in der Methode `onlyStones()`

überprüft, die alle mitfahrenden Komponenten durch Schleifen auf Steine untersucht und einen entsprechenden `boolean` zurückgibt.

Werden alle Bedingungen eingehalten, dann wird in `createRide()` die Fahrt erstellt. Dafür werden in der Liste mit *allen* Komponenten die Indizes der mitfahrenden Komponenten durch `indexOf()` in Erfahrung gebracht und einer neuen Instanz von `Ride` übergeben. Die entstehende Fahrt wird der zuvor erstellten Liste von Fahrten hinzugefügt. Nachdem alle Fahrten überprüft und evtl. erstellt wurden, wird diese Liste von `generate()` zurückgegeben, damit danach in einem `SolutionFinder` eine kürzeste Lösung gesucht werden kann.

2.4 Suchen einer Lösung

Der `SolutionFinder`, der die Breitensuche macht, um eine kürzeste Lösung zu finden, bekommt die Liste mit allen möglichen Fahrten, die vom `PossibleRidesGenerator` erstellt wurde, und die `ComponentList` übergeben. Im Konstruktor wird der Status erstellt, der alle Startpositionen beinhaltet. Danach wird die Warteschlange für die Breitensuche erstellt, dafür wird die Klasse `LinkedList` aus dem Package `java.util` verwendet. Der Warteschlange wird mit der Methode `add()` eine neue Instanz der Klasse `State` hinzugefügt, die die Liste mit allen Endpositionen beinhaltet. Diese Liste wird von der Instanz der Klasse `ComponentList` abgefragt, da sie diese Liste schon im Konstruktor erstellt. Außerdem wird beim Erstellen des `SolutionFinder`s auch eine Liste mit allen schon besuchten Status erstellt.

In `execute()` wird mit einer `while`-Schleife, welche beendet wird, wenn kein Objekt mehr in der Warteschlange enthalten ist, immer wieder der vorderste Status mit der Methode `poll()` aus der Warteschlange geholt. Ist der Status noch nicht in der Liste mit allen schon besuchten Status enthalten, wird fortgefahren. Es wird ein Übergang (`Transition`) vom aktuellen Status aus der Warteschleife zum Status mit allen Startpositionen erstellt. Ist der möglich, werden die Fahrten, die zum aktuellen Status führen, zurückgegeben, da die Lösung gefunden wurde. Ansonsten wird der Status der Liste mit allen schon besuchten Status hinzugefügt und alle möglichen Fahrten werden in einer Schleife durchlaufen.

Um zu testen, ob eine Fahrt mit dem aktuellen Status funktioniert, wird wieder ein Übergang zwischen dem aktuellen Status und demjenigen Status erstellt, der für die Fahrt benötigt wird. Wie genau dieser Status aussieht, wird unter „Erstellen eines für eine Fahrt benötigten Status“ beschrieben, jedoch bekommen die Komponenten, die an der Fahrt nicht teilnehmen, hier vorläufig einfach die Position `null`. Sie werden später in der Klasse `Transition` korrigiert und an die benötigte Position angepasst. Bei jedem Status, den der Übergang erstellt, wird die aktuelle Fahrt rückwärts durchgeführt, indem einfach in der Methode `back()` die entsprechenden Positionen des Status auf die bei der Fahrt rückwärts entstehenden Positionen gesetzt werden. Danach wird dieser Status der Warteschlange hinzugefügt und es geht mit dem nächsten Status weiter.

2.5 Übergang

Der Übergang (`Transition`) wird durch eine Methode namens `execute()` ausgeführt. Darin werden die Positionen aller Komponenten einfach durchlaufen. Wenn eine benötigte Position wie oben beschrieben `null` ist, dann wird die zugehörige Komponente für die Fahrt nicht verwendet und die benötigte Position wird an die aktuelle angepasst. Danach wird in `changePosition()` überprüft, ob der Übergang für diese Komponente möglich ist. Bei Erfolg gibt diese Methode `true`, ansonsten `false` zurück. Merkt sie, dass der Übergang zwar möglich ist, jedoch eine Person benötigt wird, weil die Positionen zwar auf derselben Seite, jedoch nicht identisch sind, wird eine der Instanzvariablen `personNeededTop` bzw. `personNeededBottom`, abhängig von der Seite, auf der sich die Komponente befindet, auf `true` gesetzt. Hat diese Methode `false` zurückgegeben, so ist der Übergang auf keinen Fall möglich und `execute()` gibt eine leere Liste zurück. Ansonsten wird innerhalb der Schleife noch überprüft, ob die Komponente eine Person ist. Wenn ja, wird `personIsTop` oder `personIsBottom`, abhängig von der Seite, auf der sie sich befindet, auf `true` gesetzt.

Hat die Schleife alle Komponenten abgearbeitet, dann kann mit Überprüfungen fortgefahren werden.

- Werden oben *und* unten Personen benötigt, es ist jedoch auf keiner Seite eine Person (was durch `personNeededTop`, `personNeededBottom`, `personIsTop` und `personIsBottom` überprüft wird), dann wird die in `personsNeeded()` entstehende Liste, die Status enthält, zurückgegeben. Diese Methode arbeitet mit verschachtelten Schleifen, durch die alle Komponenten miteinander kombiniert werden. Wenn zwei verschiedene Personen eine unbekannte Seite haben, wird ein neuer Status anhand des Status mit den benötigten Positionen erstellt, wobei die benötigten Positionen der beiden Personen auf `TOP_FREE` und `BOTTOM_FREE` gesetzt werden. Sind alle Kombinationen fertig, wird die Liste zurückgegeben. Dabei kann es natürlich vorkommen, dass es nicht möglich ist, zwei Personen zu finden, die unbekannte Seiten haben, in einem solchen Fall wird logischerweise einfach die leere Liste zurückgegeben, was bedeutet, dass der Übergang nicht möglich ist.

- Wird nur auf einer Seite eine Person benötigt, wo keine ist, wird die Methode `personNeeded()` aufgerufen, sie nimmt eine Position als Parameter. Wenn oben jemand benötigt wird, ist dies `TOP_FREE`, wenn jemand unten benötigt wird, wird `BOTTOM_FREE` übergeben. Diese Methode arbeitet genauso wie `personsNeeded()`. Der Unterschied besteht darin, dass nur eine Person in nur einer Schleife gesucht wird, die eine unbekannte Seite hat.

- Ansonsten ist überall eine Person, wo eine benötigt wird. Es wird einfach eine Liste zurückgegeben, der der Status mit den benötigten Positionen hinzugefügt wurde.

2.6 `Position`

Die Elemente des Enums `Position` werden in der Liste aller Klassen aufgezählt. Sie stellen aber alle noch drei Methoden zur Verfügung, die im gesamten Programm verwendet

werden.

Um die Position außerhalb des Korbes einer anderen Position abzurufen, wird `free()` verwendet. Diese Methode gibt in Abhängigkeit von der ursprünglichen Position `TOP_FREE`, `BOTTOM_FREE` oder bei unbekannter Seite `FREE` zurück.

Um die Seite einer Position abzufragen, wird `simpleSide()` verwendet. Diese Methode gibt `TOP`, `BOTTOM` oder bei unbekannter Seite `UNKNOWN` zurück.

Mit `simpleBasket()` wird abgefragt, ob die Position im Korb oder außerhalb ist. Es gibt die Rückgabewerte `FREE`, `BASKET` und `UNKNOWN`.

2.7 Ausgabe der Lösung

In jeder Zeile der Lösung steht eine Fahrt. Eine Fahrt ist in zwei Teile geteilt, die durch ein Pipe-Symbol getrennt werden. Auf der linken Seite stehen die Indizes der Komponenten, die runterfahren, auf der rechten Seite die der Komponenten, die hochfahren, z. B. „`[0] | [1, 3]`“.

Wie das Eingeben der Komponenten funktioniert, wird beim Start des Programmes erklärt.

3 Beispiele

Die unwichtigen Teile der Programmausgabe werden im Folgenden weggelassen. Das Einzige, was noch zu sehen bleibt, ist die Maximaldifferenz, die Liste mit den Komponenten und die zugehörige Lösung. In der Liste mit allen Klassen kann nachgelesen werden, was die Start- und Endpositionen bedeuten.

Die gesamten Ein- und Ausgaben aller Beispiele sind in der Einsendung im Ordner dieser Aufgabe enthalten. Auch die Anleitung für die Eingabe der Komponenten, welche bei Programmstart erscheint, ist dort zu finden.

3.1 Beispiel 1

Dies ist die erste Beispielaufgabe von der Website vom BwInf. Sie ist auch im Aufgabentext zu finden. Daran kann man sehen, dass für die Lösung 10 Fahrten benötigt werden. Es gibt übrigens noch zahlreiche andere Lösungen für diese Aufgabe, indem man die einzelnen Fahrten noch etwas anders anordnet. Das Programm soll aber lediglich *eine* kürzeste Lösung raussuchen. Ein gewöhnlicher Computer brauchte ca. 10 Millisekunden für die Berechnung der Lösung.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 195 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Person - Gewicht: 105 - Startposition: TOP_FREE - Endposition:
    BOTTOM
2 - Person - Gewicht: 90 - Startposition: TOP_FREE - Endposition:
    BOTTOM
```

```
3 - Stein - Gewicht: 75 - Startposition: TOP_FREE - Endposition:
    UNKNOWN

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 10 Fahrten benötigt.

[3] | []
[2] | [3]
[3] | []
[1] | [2]
[2] | [3]
[3] | []
[0] | [1, 3]
[3] | []
[1] | [2]
[2] | [3]
```

3.2 Beispiel 2

Dies ist die zweite Beispielaufgabe. Ein gewöhnlicher Computer benötigte ca. 130 Millisekunden für die Berechnung.

```
Es wird die Maximaldifferenz 1 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Stein - Gewicht: 1 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
1 - Person - Gewicht: 2 - Startposition: TOP_FREE - Endposition:
    BOTTOM
2 - Stein - Gewicht: 4 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
3 - Person - Gewicht: 8 - Startposition: TOP_FREE - Endposition:
    BOTTOM
4 - Stein - Gewicht: 16 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
5 - Person - Gewicht: 32 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Stein - Gewicht: 64 - Startposition: TOP_FREE - Endposition:
    UNKNOWN

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 14 Fahrten benötigt.

[0] | []
[1] | [0]
```

```
[0, 2, 4] | []
[3] | [0, 1, 2]
[0, 2] | []
[1] | [0]
[0] | []
[5] | [0, 1, 2, 3, 4]
[0, 2] | []
[1] | [0]
[0] | []
[3] | [0, 1, 2]
[0, 2, 4, 6] | []
[1] | [0]
```

3.3 Beispiel 3

Dies ist die dritte Beispielaufgabe. Da sie insgesamt 511 Fahrten für eine kürzeste Lösung benötigt, wird die Lösung nicht in der Dokumentation gezeigt, sondern kann nur unter den Beispieldateien im Ordner dieser Aufgabe gefunden werden. Ein gewöhnlicher Computer benötigte ca. 100 Millisekunden.

```
Es wird die Maximaldifferenz 1 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 1 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Person - Gewicht: 2 - Startposition: TOP_FREE - Endposition:
    BOTTOM
2 - Person - Gewicht: 4 - Startposition: TOP_FREE - Endposition:
    BOTTOM
3 - Person - Gewicht: 8 - Startposition: TOP_FREE - Endposition:
    BOTTOM
4 - Person - Gewicht: 16 - Startposition: TOP_FREE - Endposition:
    BOTTOM
5 - Person - Gewicht: 32 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Person - Gewicht: 64 - Startposition: TOP_FREE - Endposition:
    BOTTOM
7 - Person - Gewicht: 128 - Startposition: TOP_FREE - Endposition:
    BOTTOM
8 - Person - Gewicht: 256 - Startposition: TOP_FREE - Endposition:
    BOTTOM

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 511 Fahrten benötigt.
...
```

3.4 Beispiel 4

Die 4. Beispielaufgabe von der Website. Ein gewöhnlicher Computer benötigte ca. 1500 Millisekunden.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 116 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Person - Gewicht: 231 - Startposition: TOP_FREE - Endposition:
    BOTTOM
2 - Stein - Gewicht: 50 - Startposition: BOTTOM_FREE - Endposition:
    UNKNOWN
3 - Person - Gewicht: 55 - Startposition: TOP_FREE - Endposition:
    BOTTOM
4 - Person - Gewicht: 101 - Startposition: BOTTOM_FREE - Endposition:
     BOTTOM
5 - Person - Gewicht: 185 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Person - Gewicht: 211 - Startposition: BOTTOM_FREE - Endposition:
     BOTTOM
7 - Person - Gewicht: 200 - Startposition: BOTTOM_FREE - Endposition:
     BOTTOM
8 - Person - Gewicht: 224 - Startposition: TOP_FREE - Endposition:
    BOTTOM

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 27 Fahrten benötigt.

[1, 3, 5] | [2, 6, 7]
[2] | []
[0] | [2, 3]
[2] | []
[3, 6, 7] | [0, 2, 4, 5]
[2] | []
[0] | [2, 3]
[2] | []
[3, 4, 8] | [0, 2, 7]
[2] | []
[0] | [2, 3]
[2] | []
[5, 7] | [2, 4, 8]
[2] | []
[3, 4, 8] | [0, 2, 7]
[2] | []
[0] | [2, 3]
[2] | []
```

```
[3] | [2]
[2] | []
[7] | [5]
[5] | [0, 3]
[3] | [2]
[2] | []
[0] | [2, 3]
[2] | []
[3] | [2]
```

3.5 Beispiel 5

Dies ist die Lösung für die 5. Beispielaufgabe. Ein gewöhnlicher Computer benötigte ca. 4 Sekunden. Für diese Aufgabe wird mit Abstand am meisten Zeit in Anspruch genommen.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 195 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Person - Gewicht: 105 - Startposition: TOP_FREE - Endposition:
    BOTTOM
2 - Person - Gewicht: 90 - Startposition: TOP_FREE - Endposition:
    BOTTOM
3 - Stein - Gewicht: 75 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
4 - Person - Gewicht: 137 - Startposition: TOP_FREE - Endposition:
    BOTTOM
5 - Person - Gewicht: 55 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Person - Gewicht: 101 - Startposition: BOTTOM_FREE - Endposition:
     BOTTOM
7 - Person - Gewicht: 185 - Startposition: TOP_FREE - Endposition:
    BOTTOM
8 - Person - Gewicht: 199 - Startposition: TOP_FREE - Endposition:
    BOTTOM

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 23 Fahrten benötigt.

[3] | []
[2] | [3]
[3] | []
[7] | [3, 6]
[3] | []
[1, 5, 8] | [2, 3, 7]
```

```
[3] | []
[2] | [3]
[3] | []
[6, 7] | [3, 8]
[3] | []
[0] | [1, 3]
[3] | []
[1, 4, 8] | [0, 3, 5, 6]
[3] | []
[0] | [1, 3]
[3] | []
[1, 5, 6] | [3, 7]
[3] | []
[7] | [3, 6]
[3] | []
[6] | [2]
[2] | [3]
```

3.6 Beispiel 6

Das 6. Beispiel, ist die einzige Beispielaufgabe, die auf der Website zu finden ist, zu der es keine Lösung gibt. Die Berechnung dauerte ca. 25 Millisekunden auf einem gewöhnlichen Computer.

```
Es wird die Maximaldifferenz 20 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 109 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Stein - Gewicht: 120 - Startposition: BOTTOM_FREE - Endposition:
    UNKNOWN
2 - Person - Gewicht: 156 - Startposition: TOP_FREE - Endposition:
    BOTTOM
3 - Person - Gewicht: 55 - Startposition: TOP_FREE - Endposition:
    BOTTOM
4 - Person - Gewicht: 149 - Startposition: TOP_FREE - Endposition:
    BOTTOM
5 - Person - Gewicht: 185 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Person - Gewicht: 98 - Startposition: BOTTOM_FREE - Endposition:
    BOTTOM

================================================================

Es gibt keine Lösung.
```

3.7 Beispiel 7

Das 7. und damit das letzte Beispiel der Website entspricht dem 6. Beispiel bis auf das Gewicht der letzten Person. Statt 98 wird hier 85 verwendet. Hier gibt es eine Lösung. Die Berechnung benötigte ca. 90 Millisekunden.

```
Es wird die Maximaldifferenz 20 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 109 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Stein - Gewicht: 120 - Startposition: BOTTOM_FREE - Endposition:
    UNKNOWN
2 - Person - Gewicht: 156 - Startposition: TOP_FREE - Endposition:
    BOTTOM
3 - Person - Gewicht: 55 - Startposition: TOP_FREE - Endposition:
    BOTTOM
4 - Person - Gewicht: 149 - Startposition: TOP_FREE - Endposition:
    BOTTOM
5 - Person - Gewicht: 185 - Startposition: TOP_FREE - Endposition:
    BOTTOM
6 - Person - Gewicht: 85 - Startposition: BOTTOM_FREE - Endposition:
    BOTTOM

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 9 Fahrten benötigt.

[2, 3] | [1, 6]
[1] | []
[5] | [1, 3]
[1] | []
[0, 3, 4] | [1, 5]
[1] | []
[5] | [1, 3]
[1] | []
[3, 6] | [1]
```

3.8 Beispiel 8

Die folgenden Beispiele gehören nicht mehr zu den Beispielaufgaben, die auf der Website vom BwInf zu finden sind. Sie sollen lediglich die Funktionsweise meiner Erweiterungen vorstellen.

In diesem Beispiel startet eine Person oben und ein Stein unten. Beide Komponenten sollen am Ende unten sein. Dieses Beispiel zeigt, dass man für die Startpositionen beliebige spezifische Positionen verwenden kann. Das wird hier nämlich benötigt, denn damit die Person nach unten fahren kann, muss der Stein unten im Korb sein. Durch die

Erweiterung ist es möglich, dies direkt als Startposition angeben zu können. Außerdem kann man an diesem Beispiel erkennen, dass auch die Endpositionen der Komponenten verändert werden können. So soll hier z. B. der Stein zum Schluss auch unten sein. Das ist ohne Probleme möglich, denn nachdem die Person nach unten und der Stein nach oben gefahren sind, kann der Stein ganz einfach wieder nach unten fahren, ohne dass oben eine weitere Person benötigt wird, denn er kann einfach im Korb bleiben. Ein gewöhnlicher Computer benötigte ca. 6 Millisekunden für die Berechnung.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 50 - Startposition: TOP_FREE - Endposition:
    BOTTOM
1 - Stein - Gewicht: 35 - Startposition: BOTTOM_BASKET - Endposition:
     BOTTOM

================================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 2 Fahrten benötigt.

[0] | [1]
[1] | []
```

3.9 Beispiel 9

Hier wird eine Endposition für den Stein angegeben. Er soll am Ende unten sein, jedoch außerhalb des Korbes. Es gibt also keine Lösung, da eine Person benötigt wird, die ihn aus dem Korb raushebt, obwohl der Stein von alleine runter fahren kann. Ein gewöhnlicher Computer benötigte für die Berechnung ca. 4 Millisekunden.

```
Es wird die Maximaldifferenz 1 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Stein - Gewicht: 50 - Startposition: TOP_BASKET - Endposition:
    BOTTOM_FREE

================================================================================

Es gibt keine Lösung.
```

3.10 Beispiel 10

Hier das Beispiel, welches unter „Lösungsidee“ verwendet wurde. Dabei weichen die Endpositionen wieder von den Standard-Endpositionen ab. Es wurden für die Berechnung ca. 8 Millisekunden auf einem gewöhnlichen Computer benötigt.

```
Es wird die Maximaldifferenz 5 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 20 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
1 - Person - Gewicht: 15 - Startposition: BOTTOM_FREE - Endposition:
    TOP
2 - Person - Gewicht: 80 - Startposition: TOP_FREE - Endposition:
    BOTTOM
3 - Stein - Gewicht: 75 - Startposition: TOP_FREE - Endposition:
    BOTTOM

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 4 Fahrten benötigt.

[0] | [1]
[3] | []
[2] | [3]
[3] | []
```

3.11 Beispiel 11

Hier werden noch einmal die Endpositionen abgeändert. Der Stein soll am Ende irgendwo außerhalb der Körbe sein, die Person unten. Die Lösung ist ganz einfach: der Stein fährt runter. Dort hebt ihn dann die Person aus dem Korb und beide sind dort, wo sie sein sollen. Ein gewöhnlicher Computer benötigte ca. 8 Millisekunden.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Stein - Gewicht: 50 - Startposition: TOP_BASKET - Endposition:
    FREE
1 - Person - Gewicht: 100 - Startposition: BOTTOM_FREE - Endposition:
     BOTTOM

================================================================

Lösung suchen...

================================================================

Eine kürzeste Lösung wurde gefunden.
Dafür werden 1 Fahrten benötigt.

[0] | []
```

3.12 Beispiel 12

Beim letzten Beispiel muss wegen einem Sonderfall die Position der Person, noch während eines Überganges, von p_e^e auf p_f^u gesetzt werden, da der Stein unten eine Person benötigt, die ihn aus dem Korb hebt, jedoch keine anwesend ist. Es wäre nämlich nicht möglich, wenn der Stein einfach nur runterfährt, denn der Stein soll am Ende unten, außerhalb des Korbes sein. Ein gewöhnlicher Computer benötigte ca. 7 Millisekunden für die Berechnung.

```
Es wird die Maximaldifferenz 15 verwendet.

Für folgende Komponenten wird eine Lösung gesucht:
0 - Person - Gewicht: 50 - Startposition: TOP_FREE - Endposition:
    UNKNOWN
1 - Stein - Gewicht: 35 - Startposition: TOP_FREE - Endposition:
    BOTTOM_FREE

================================================================


Eine kürzeste Lösung wurde gefunden.
Dafür werden 3 Fahrten benötigt.

[1] | []
[0] | [1]
[1] | []
```

4 Quelltext

Listing 1: `Position`

```
public enum Position {

    /**
     * Oben, außerhalb des Korbes.
     */
    TOP_FREE {
        public Position free() { return TOP_FREE; }
        public Position simpleSide() { return TOP; }
        public Position simpleBasket() { return FREE; }
    },
    /**
     * Unten, außerhalb des Korbes.
     */
    BOTTOM_FREE {
        public Position free() { return BOTTOM_FREE; }
        public Position simpleSide() { return BOTTOM; }
        public Position simpleBasket() { return FREE; }
    },
```

```
    /**
     * Oben, im Korb.
     */
    TOP_BASKET {
        public Position free() { return TOP_FREE; }
        public Position simpleSide() { return TOP; }
        public Position simpleBasket() { return BASKET; }
    },
    /**
     * Unten, im Korb.
     */
    BOTTOM_BASKET {
        public Position free() { return BOTTOM_FREE; }
        public Position simpleSide() { return BOTTOM; }
        public Position simpleBasket() { return BASKET; }
    },
    /**
     * Oben. Es ist egal, ob die Position im Korb ist, oder nicht.
     */
    TOP {
        public Position free() { return TOP_FREE; }
        public Position simpleSide() { return TOP; }
        public Position simpleBasket() { return UNKNOWN; }
    },
    /**
     * Unten. Es ist egal, ob die Position im Korb ist, oder nicht.
     */
    BOTTOM {
        public Position free() { return BOTTOM_FREE; }
        public Position simpleSide() { return BOTTOM; }
        public Position simpleBasket() { return UNKNOWN; }
    },
    /**
     * Außerhalb des Korbes. Es ist egal, ob die Position oben oder unten ist.
     */
    FREE {
        public Position free() { return FREE; }
        public Position simpleSide() { return UNKNOWN; }
        public Position simpleBasket() { return FREE; }
    },
    /**
     * Im Korb. Es ist egal, ob die Position oben oder unten ist.
     */
    BASKET {
        public Position free() { return FREE; }
        public Position simpleSide() { return UNKNOWN; }
        public Position simpleBasket() { return BASKET; }
    },
    /**
```

```
     * Die Position ist unbekannt.
     */
    UNKNOWN {
        public Position free() { return FREE; }
        public Position simpleSide() { return UNKNOWN; }
        public Position simpleBasket() { return UNKNOWN; }
    };

    /**
     * @return die Position auf derselben Seite, jedoch außerhalb des Korbes
     */
    public abstract Position free();

    /**
     * @return die Seite, also TOP, BOTTOM oder UNKNOWN
     */
    public abstract Position simpleSide();

    /**
     * @return ob sich die Position im Korb befindet, oder nicht, also
     * BASKET, FREE oder UNKNOWN
     */
    public abstract Position simpleBasket();

}
```

Listing 2: Component

```
public abstract class Component {

    private int weight;
    private Position startPosition;
    private Position endPosition;

    /**
     * @param weight      das Gewicht der Komponente
     * @param startPosition die Startposition
     * @param endPosition die Endposition
     */
    public Component(int weight, Position startPosition, Position
            endPosition) {
        this.weight = weight;
        this.startPosition = startPosition;
        this.endPosition = endPosition;
    }

    public int getWeight() {
        return weight;
    }
```

```
    public Position startPosition() {
        return startPosition;
    }

    public Position endPosition() {
        return endPosition;
    }

}
```

Listing 3: `Person`

```
public class Person extends Component {

    public Person(int weight, Position startPosition, Position endPosition) {
        super(weight, startPosition, endPosition);
    }

}
```

Listing 4: `Stone`

```
public class Stone extends Component {

    public Stone(int weight, Position startPosition, Position endPosition) {
        super(weight, startPosition, endPosition);
    }

}
```

Listing 5: `ComponentList`

```
public class ComponentList {

    private List<Component> components;
    private List<Position> onStart; // Startpositionen
    private List<Position> onEnd; // Endpositionen

    public ComponentList(List<Component> components) {
        this.components = components;

        onStart = new ArrayList<>();
        onEnd = new ArrayList<>();

        for (Component component : components) {
            onStart.add(component.startPosition());
            onEnd.add(component.endPosition());
        }
    }
```

```
    public List<Component> getComponents() {
        return components;
    }

    public List<Position> onStart() {
        return onStart;
    }

    public List<Position> onEnd() {
        return onEnd;
    }

    /**
     * Überprüft, ob die Komponente des angegebenen Index eine Person ist.
     */
    public boolean isPerson(int index) {
        return components.get(index) instanceof Person;
    }

}
```

Listing 6: `Ride`

```
public class Ride {

    private List<Integer> top, bottom;

    /**
     * @param top die Komponenten, die anfangs oben sind und nach unten fahren
     * @param bottom die Komponenten, die anfangs unten sind und nach oben
     *               fahren
     */
    public Ride(List<Integer> top, List<Integer> bottom) {
        this.top = top;
        this.bottom = bottom;
    }

    public List<Integer> getTop() {
        return top;
    }

    public List<Integer> getBottom() {
        return bottom;
    }

    @Override
    public String toString() {
        return top + " | " + bottom;
    }
```

```
}
```

Listing 7: PossibleRidesGenerator

```
public class PossibleRidesGenerator {

    private ComponentList componentList;
    private int maxWeightDifference;

    /**
     * @param componentList    die Liste mit allen Komponenten
     * @param maxWeightDifference der maximale Unterschied zwischen den
     *                           Gewichten des oberen und des unteren Korbes
     */
    public PossibleRidesGenerator(ComponentList componentList, int
            maxWeightDifference) {
        this.componentList = componentList;
        this.maxWeightDifference = maxWeightDifference;
    }

    /**
     * Generiert eine Liste mit allen möglichen Fahrten.
     */
    public List<Ride> generate() {

        // Die Potenzmenge aller Komponenten als Map erstellen und die
        // Gewichte (Werte der Map) aller Teilmengen
        // (Schlüssel der Map) berechnen.
        Map<List<Component>, Integer> powerSet =
                powerSet(componentList.getComponents());

        // Alle möglichen Fahrten erstellen und zurückgeben.
        return generatePossibleRides(powerSet, componentList.getComponents());

    }

    /**
     * Generiert die Potenzmenge (engl. "power set") der Liste mit allen
     * Komponenten als Map. Die Map beinhaltet als Schlüssel eine Teilmenge
     * und direkt als Wert das zugehörige Gewicht, damit dieses später
     * nicht unnötig mehrmals berechnet werden muss.
     *
     * @param components alle Komponenten
     * @return die Potenzmenge
     */
    private Map<List<Component>, Integer> powerSet(List<Component>
                                                           components) {
        Map<List<Component>, Integer> powerSet = new HashMap<>();
        powerSet.put(new ArrayList<Component>(), 0);
```

```
        // Für jede Komponente neue Teilmengen erstellen, wobei jede
        // bisherige Teilmenge genommen und einer Kopie davon die
        // Komponente hinzugefügt wird.
        for (Component component : components) {

            // newSubList: die neu entstehenden Teilmengen. Sie können der
            // Potenzmenge nicht direkt hinzugefügt werden, da durch das
            // Hinzufügen während einer Iteration der Fehler java.util
            // .ConcurrentModificationException auftritt.
            Map<List<Component>, Integer> newSubLists = new HashMap<>();

            // Alle bisherigen Teilmengen durchlaufen
            for (List<Component> subList : powerSet.keySet()) {

                // Neue Teilmenge erstellen, wobei die aktuelle Komponente
                // einer Kopie von subList hinzugefügt wird.
                List<Component> newSubList = new ArrayList<>(subList);
                newSubList.add(component);

                // newSubList der Map newSubLists hinzufügen, wobei das
                // Gewicht der Teilmenge direkt berechnet wird.
                newSubLists.put(newSubList, weight(newSubList));
            }

            powerSet.putAll(newSubLists);
        }

        return powerSet;
    }

    /**
     * Berechnet das Gesamtgewicht der Komponenten in der angegebenen Liste.
     */
    private int weight(List<Component> components) {

        int weight = 0;

        for (Component component : components)
            weight += component.getWeight();

        return weight;
    }

    /**
     * Generiert alle möglichen Fahrten mithilfe der angegebenen
     * Potenzmenge der Komponenten und deren Gewichten.
     *
     * @param powerSet die Potenzmenge aller Komponenten
     * @param components alle Komponenten
```

```
     * @return alle möglichen Fahrten
     */
    private List<Ride> generatePossibleRides(Map<List<Component>,
            Integer> powerSet, List<Component> components) {

        // possibleRides: Liste mit allen möglichen Fahrten
        List<Ride> possibleRides = new ArrayList<>();

        // Alle Teilmengen als top bzw. bottom miteinander kombinieren.
        // Sollten diese so passen, dass eine Fahrt mit ihnen erstellt
        // werden kann, die alle Bedingungen einhält, wird sie der Liste
        // possibleRides hinzugefügt.
        for (Map.Entry<List<Component>, Integer> top : powerSet.entrySet()) {
            for (Map.Entry<List<Component>, Integer> bottom :
                    powerSet.entrySet()) {

                // Teilmengen aus top bzw. bottom auslesen (sie sind als
                // Schlüssel gespeichert).
                List<Component> topComponents = top.getKey();
                List<Component> bottomComponents = bottom.getKey();

                // Eine Komponente darf nicht gleichzeitig unten und oben sein
                if (containSameComponents(topComponents, bottomComponents))
                    continue;

                // Die Differenz zwischen den Gesamtgewichten (den Werten
                // von top bzw. bottom) berechnen.
                int weightDifference = top.getValue() - bottom.getValue();

                // Die Fahrt kann nicht stattfinden, wenn die Differenz <= 0
                // ist, da der obere Korb schwerer belastet werden muss
                // (das ist nur dann der Fall, wenn weightDifference
                // positiv ist). Außerdem ist die Fahrt nicht möglich, wenn
                // die Differenz die Maximaldifferenz überschreitet und min.
                // eine Person in einer der Körben sitzt, da Personen
                // "sicher" fahren müssen.
                if (weightDifference <= 0 ||
                        (weightDifference > maxWeightDifference &&
                                !onlyStones(topComponents, bottomComponents)))
                    continue;

                // Alle Bedingungen wurden eingehalten, daher kann die
                // Fahrt erstellt und der Liste hinzugefügt werden.
                possibleRides.add(createRide(topComponents,
                        bottomComponents, components));
            }
        }

        return possibleRides;
```

```
    }

    /**
     * Überprüft, ob es eine Komponente gibt, die in beiden Listen
     * enthalten ist.
     */
    private boolean containSameComponents(List<Component> top,
                                          List<Component> bottom) {

        // Mithilfe von verschachtelten Schleifen true zurückgeben, wenn
        // eine Komponente in beiden Liste enthalten ist.
        for (Component componentTop : top)
            for (Component componentBottom : bottom)
                if (componentTop == componentBottom)
                    return true;

        return false;
    }

    /**
     * Überprüft, ob in beiden Listen nur Steine enthalten sind.
     */
    private boolean onlyStones(List<Component> top, List<Component> bottom) {

        // Wenn irgendeine Komponente eine Person ist, wird false
        // zurückgegeben.
        for (Component component : top)
            if (component instanceof Person)
                return false;
        for (Component component : bottom)
            if (component instanceof Person)
                return false;

        return true;
    }

    /**
     * Erstellt eine Fahrt anhand der angegebenen Komponenten, die nach
     * oben bzw. nach unten fahren.
     *
     * @return die Fahrt
     */
    private Ride createRide(List<Component> top, List<Component> bottom,
                            List<Component> components) {

        // Liste mit den Indizes von den Komponenten erstellen, die oben
        // bzw. unten sind.
        List<Integer> topIDs = new ArrayList<>();
        for (Component component : top)
```

```
            topIDs.add(components.indexOf(component));

        List<Integer> bottomIDs = new ArrayList<>();
        for (Component component : bottom)
            bottomIDs.add(components.indexOf(component));

        return new Ride(topIDs, bottomIDs);
    }

}
```

Listing 8: `State`

```
public class State {

    /**
     * Die Positionen der Komponenten.
     * Die Indizes dieser Positionen entsprechen denen der Komponenten der
     * ComponentList.
     */
    protected List<Position> positions;
    /**
     * Die Folge von Fahrten, die zu diesem Status führt.
     */
    protected List<Ride> order;

    /**
     * Erstellt einen neuen Status anhand der angegebenen Positionen.
     */
    public State(List<Position> positions) {
        this.positions = positions;
        this.order = new ArrayList<>();
    }

    /**
     * Erstellt einen neuen Status, indem die Positionen und die Folge von
     * Fahrten vom angegebenen Status kopiert werden.
     */
    public State(State old) {
        positions = new ArrayList<>(old.positions);
        order = new ArrayList<>(old.order);
    }

    /**
     * Erstellt einen neuen Status anhand der angegebenen Fahrt und der
     * Liste mit allen Komponenten. Der Status bezieht sich darauf, wie
     * er aussehen muss, NACHDEM die Fahrt durchgeführt wurde.
     * Dabei werden alle irrelevanten Positionen auf null gesetzt.
     */
    public State(Ride ride, ComponentList componentList) {
```

```
        positions = new ArrayList<>();

        for (int i = 0; i < componentList.getComponents().size(); i++) {
            if (ride.getBottom().contains(i))
                positions.add(Position.TOP_BASKET);
            else if (ride.getTop().contains(i))
                positions.add(Position.BOTTOM_BASKET);
            else
                positions.add(null);
        }
    }

    /**
     * Führt den angegebenen Zug rückwärts aus. Außerdem wird
     * der Zug zu der Folge mit allen Fahrten hinzugefügt.
     */
    public void back(Ride ride) {

        // Ganz vorne in order tun, da die Fahrt sozusagen vor allen zuvor
        // berechneten Fahrten ist:
        order.add(0, ride);

        // Komponenten, die nach unten fahren, bekommen die Position
        // TOP_BASKET.
        for (int top : ride.getTop())
            positions.set(top, Position.TOP_BASKET);

        // Komponenten, die nach oben fahren, bekommen die Position
        // BOTTOM_BASKET.
        for (int bottom : ride.getBottom())
            positions.set(bottom, Position.BOTTOM_BASKET);

    }

}
```

Listing 9: `Transition`

```
public class Transition {

    /**
     * Der Status, zu dem dieser Übergang führen soll.
     */
    private State neededState;
    /**
     * Der Status, wie er aktuell ist.
     */
    private State currentState;
    /**
```

```
     * Die Liste mit allen Komponenten.
     */
    private ComponentList componentList;

    /**
     * Diese Variablen besagen, ob Personen oben bzw. unten benötigt werden.
     */
    private boolean personNeededTop = false, personNeededBottom = false;
    /**
     * Diese Variablen besagen, ob min. eine Person oben bzw. unten ist.
     */
    private boolean personIsTop = false, personIsBottom = false;

    public Transition(State neededState, State currentState,
                      ComponentList componentList) {
        this.neededState = neededState;
        this.currentState = currentState;
        this.componentList = componentList;
    }

    /**
     * Überprüft den Übergang.
     *
     * @return alle Status, die nach dem Übergang möglich sind,
     * möglicherweise auch gar keine
     */
    public List<State> execute() {

        // Alle Positionen überprüfen
        for (int i = 0; i < neededState.positions.size(); i++) {

            // aktuelle und benötigte Positionen:
            Position needed = neededState.positions.get(i);
            Position current = currentState.positions.get(i);

            if (needed == null) {
                // Wenn die benötigte Position null ist, dann nimmt die
                // zugehörige Komponente nicht an der Fahrt teil. Daher ist
                // die benötigte Position außerhalb des Korbes, aber auf
                // der Seite der aktuellen Position:
                needed = current.free();

                // Auch der Status muss die neue Position bekommen:
                this.neededState.positions.set(i, needed);
            }

            if (!changePosition(current, needed))
                return new ArrayList<>(); // Eine Komponente müsste die
            // Seite ändern, also gibt es keine Lösung.
```

```
            // Wenn die Komponente eine Person ist, wird personIsTop bzw.
            // personIsBottom auf true gesetzt.
            if (componentList.isPerson(i) &&
                    needed.simpleSide() == Position.TOP)
                personIsTop = true;

            else if (componentList.isPerson(i) &&
                    needed.simpleSide() == Position.BOTTOM)
                personIsBottom = true;

        }

        // Die bisherige Folge von Fahrten kopieren und dem benötigten
        // Status hinzufügen, da dieser die Folge noch nirgends bekommen
        // hat, aber dieselbe Folge von Fahrten dorthin führt.
        this.neededState.order = new ArrayList<>(this.currentState.order);

        if ((personNeededTop && !personIsTop) &&
                (personNeededBottom && !personIsBottom)) {
            // Es werden Personen unten und oben benötigt, es ist aber
            // keine Person oben und es ist keine Person unten.
            return personsNeeded();

        } else if (personNeededTop && !personIsTop) {
            // Eine Person wird oben benötigt, es ist aber keine oben.
            return personNeeded(Position.TOP_FREE);

        } else if (personNeededBottom && !personIsBottom) {
            // Eine Person wird unten benötigt, es ist aber keine unten.
            return personNeeded(Position.BOTTOM_FREE);

        } else {
            // Der Übergang funktioniert reibungslos, also wird der Status
            // zurückgegeben.
            List<State> states = new ArrayList<>();
            states.add(this.neededState);
            return states;
        }
    }

    /**
     * Überprüft, ob eine Komponente von der Position current zu needed
     * wechseln kann.
     */
    private boolean changePosition(Position current, Position needed) {
        if (current.simpleSide() == needed.simpleSide() ||
                current.simpleSide() == Position.UNKNOWN) {
            // Auf derselben Seite oder die aktuelle Seite ist egal.
```

```
        if (current.simpleBasket() == needed.simpleBasket() ||
                current.simpleBasket() == Position.UNKNOWN) {
            // Beide Positionen sind frei bzw. im Korb oder es ist
            // egal, ob die Komponente aktuell im Korb ist. Der Wechsel
            // für diese Komponente ist auf jeden Fall möglich.
            return true;

        } else {
            // Die Positionen sind nicht beide frei bzw. im Korb, also
            // wird auf dieser Seite eine Person benötigt.
            if (needed.simpleSide() == Position.TOP)
                personNeededTop = true;
            else personNeededBottom = true;
            return true;
        }

    } else // Auf unterschiedlichen Seiten -> nicht möglich
        return false;
}

/**
 * Eine Person mit unbekannter Seite wird gesucht. Alle dabei
 * entstehenden Status werden zurückgegeben.
 */
private List<State> personNeeded(Position position) {

    List<State> states = new ArrayList<>();

    for (int i = 0; i < neededState.positions.size(); i++) {
        if (componentList.isPerson(i) && neededState.positions.get(i)
                .simpleSide() == Position.UNKNOWN) {

            // Status erstellen, die Position der Person festlegen und
            // den Status der Liste hinzufügen.
            State state = new State(neededState);
            state.positions.set(i, position);
            states.add(state);
        }
    }

    return states;
}

/**
 * Zwei Personen mit unbekannten Seiten werden gesucht. Alle dabei
 * entstehenden Status werden zurückgegeben.
 */
private List<State> personsNeeded() {
```

```
        List<State> states = new ArrayList<>();

        for (int i = 0; i < neededState.positions.size(); i++) {
            if (componentList.isPerson(i) &&
                    neededState.positions.get(i) == Position.UNKNOWN) {

                // Komponente wird mit allen anderen Komponenten kombiniert:
                for (int j = 0; j < neededState.positions.size(); j++) {

                    if (componentList.isPerson(j) && neededState.positions
                            .get(j).simpleSide() == Position.UNKNOWN
                            && i != j) {

                        // Status erstellen, die Positionen der Personen
                        // festlegen und den Status der Liste hinzufügen.
                        State state = new State(neededState);
                        state.positions.set(i, Position.TOP_FREE);
                        state.positions.set(j, Position.BOTTOM_FREE);
                        states.add(state);
                    }
                }
            }
        }

        return states;
    }

}
```

Listing 10: `SolutionFinder`

```
public class SolutionFinder {


    /**
     * Schon besuchte Status. Kommt die Breitensuche bei einem Status
     * an, der schon besucht wurde, wird dort nicht weitergesucht.
     */
    private List<List<Position>> found = new ArrayList<>();

    /**
     * Der "Startstatus". Kommt die Breitensuche hier an, so wurde eine
     * Lösung gefunden.
     */
    private State startState;

    private Queue<State> queue; // Warteschlange der Breitensuche
    private List<Ride> possibleRides; // alle möglichen Fahrten
    private ComponentList componentList; // alle Komponenten
```

```
public SolutionFinder(List<Ride> possibleRides, ComponentList
        componentList) {
    this.possibleRides = possibleRides;
    this.componentList = componentList;

    // Startstatus anhand der Startpositionen erstellen
    startState = new State(componentList.onStart());

    // Warteschlange erstellen und den Endstatus einfügen, bei dem die
    // Breitensuche anfängt zu suchen.
    queue = new LinkedList<>();
    queue.add(new State(componentList.onEnd()));
}

/**
 * Führt die Breitensuche aus.
 *
 * @return {@code null}, wenn keine Lösung gefunden wurde, ansonsten
 * eine kürzeste Lösung
 */
public List<Ride> execute() {

    while (!queue.isEmpty()) {

        // Erstes Element der Warteschlange entnehmen:
        State state = queue.poll();

        // Wenn der Status schon "besucht" wurde, muss hier nicht
        // weitergemacht werden.
        if (!found.contains(state.positions)) {

            // Es wurde eine Lösung gefunden, wenn der Übergang vom
            // aktuellen Status zu den Startpositionen möglich ist.
            if (new Transition(startState, state, componentList)
                    .execute().size() > 0)
                return state.order;

            // Status wird jetzt "besucht".
            found.add(state.positions);

            // Alle möglichen Fahrten durchprobieren:
            for (Ride ride : possibleRides) {

                // rideState: Der Status, der gebraucht wird, damit die
                // Fahrt durchgeführt werden kann.
                State rideState = new State(ride, componentList);

                // Bei allen möglichen Status, die durch den Übergang
```

```
                    // entstehen können, die Fahrt rückwärts durchführen
                    // und das Resultat der Warteschlange hinzufügen.
                    for (State newState : new Transition(rideState, state,
                            componentList).execute()) {
                        newState.back(ride);
                        queue.add(newState);
                    }

                }
            }
        }

        // Die Breitensuche hat keine Lösung gefunden, also null zurückgeben.
        return null;
    }

}
```

Listing 11: `Seilschaften`

```
public class Seilschaften {

    public static void main(String[] args) {
        new Seilschaften().start();
    }

    private void start() {

        // ...
        // difference ist die eingelesene Maximaldifferenz.
        // componentList ist die eingelesene Liste mit den Komponenten.

        // Alle möglichen Fahrten generieren:
        PossibleRidesGenerator possibleRidesGenerator =
                new PossibleRidesGenerator(componentList, difference);
        List<Ride> possibleRides = possibleRidesGenerator.generate();

        // Kürzeste Lösung finden:
        SolutionFinder solutionFinder =
                new SolutionFinder(possibleRides, componentList);
        List<Ride> order = solutionFinder.execute();

        // Lösung (wenn vorhanden) ausgeben ...

    }
}
```