Kapitel 2
Suchaufgaben

»Suche« ist ein so breit gefächerter Begriff, dass dieses ganze Buch *Klassische Suchaufgaben in Java* heißen könnte. In diesem Kapitel geht es um Kernalgorithmen für die Suche, die für das Programmieren wichtig sind. Trotz seiner deklaratorischen Überschrift erhebt es keinen Anspruch auf Vollständigkeit.

2.1 DNA-Suche

Gene werden in Computerprogrammen üblicherweise als Abfolgen der Zeichen *A*, *C*, *G* und *T* dargestellt. Jeder Buchstabe steht für ein *Nukleotid*, und die Kombination aus drei Nukleotiden wird *Codon* genannt. Dies wird in Abbildung 2.1 dargestellt. Ein Codon codiert für eine bestimmte Aminosäure, die zusammen mit anderen Aminosäuren ein *Protein* bilden kann. Eine klassische Aufgabe in der Bioinformatik besteht darin, ein bestimmtes Codon innerhalb eines Gens zu finden.

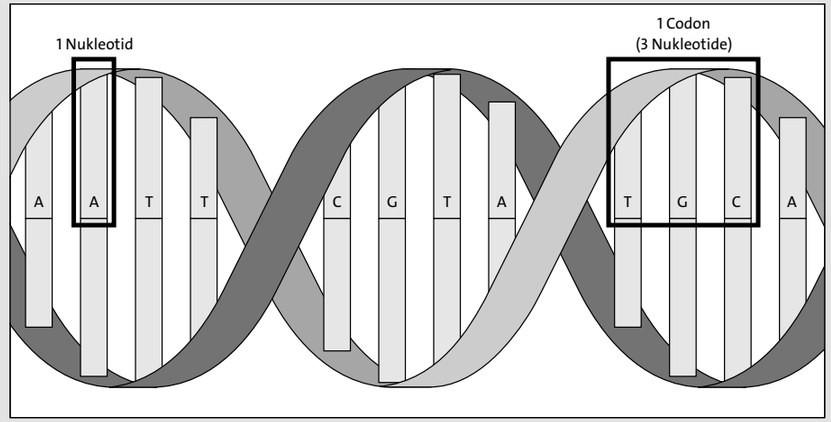

Abbildung 2.1 Ein Nukleotid wird durch einen der Buchstaben A, C, G oder T dargestellt. Ein Codon besteht aus drei Nukleotiden und ein Gen aus mehreren Codons.

2.1.1 DNA speichern

Wir können ein Nukleotid als einfaches `enum` mit vier Fällen darstellen.

```
package chapter2;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Gene {

    public enum Nucleotide {
        A, C, G, T
    }
```

Listing 2.1 Gene.java

Codons lassen sich als Kombination von drei `Nucleotide`-Objekten definieren. Der Konstruktor der `Codon`-Klasse wandelt einen `String` aus drei Buchstaben in ein `Codon` um. Zur Implementierung von Suchmethoden müssen wir in der Lage sein, zwei verschiedene `Codons` miteinander zu vergleichen. Java verfügt hierfür bereits über die Schnittstelle `Comparable`.

Die Implementierung der Schnittstelle `Comparable` erfordert die Konstruktion einer Methode `compareTo()`. `compareTo()` sollte eine negative Zahl zurückgeben, wenn das betreffende Element kleiner ist als das zu vergleichende Element, eine Null, wenn die beiden Elemente gleich sind, und eine positive Zahl, wenn das Element größer ist als das zu vergleichende Element. In der Praxis brauchen Sie dies oft nicht selbst zu implementieren und können stattdessen auf das eingebaute Java-Interface `Comparator` zurückgreifen, wie auch wir es im folgenden Beispiel tun. In diesem Beispiel wird das `Codon` zuerst anhand seines ersten `Nucleotids` mit einem anderen Codon verglichen und danach, falls die ersten beiden gleich sind, anhand seines zweiten und schließlich anhand seines dritten, falls auch die zweiten gleich sind. Die Verkettung erfolgt mit `thenComparing()`.

```
    public static class Codon implements Comparable<Codon> {
        public final Nucleotide first, second, third;
        private final Comparator<Codon> comparator =
     Comparator.comparing((Codon c) -> c.first)
                .thenComparing((Codon c) -> c.second)
                .thenComparing((Codon c) -> c.third);
```



```
        public Codon(String codonStr) {
            first = Nucleotide.valueOf(codonStr.substring(0, 1));
            second = Nucleotide.valueOf(codonStr.substring(1, 2));
            third = Nucleotide.valueOf(codonStr.substring(2, 3));
        }

        @Override
        public int compareTo(Codon other) {
            // first wird zuerst verglichen, dann second usw.
            // D. h., first hat Vorrang vor second
            // und second vor third
            return comparator.compare(this, other);
        }
    }
```

Listing 2.2 Gene.java (Fortsetzung)

Hinweis

`Codon` ist eine statische Klasse. `static` gekennzeichnete verschachtelte Klassen können ohne Rücksicht auf ihre umschließende Klasse instanziiert werden (Sie benötigen keine Instanz der umschließenden Klasse, um eine Instanz einer statischen geschachtelten Klasse zu erzeugen), aber sie können auf keine der Instanzvariablen ihrer umschließenden Klasse verweisen. Dies ist sinnvoll für Klassen, die vor allem aus organisatorischen und nicht aus logistischen Gründen als verschachtelte Klassen definiert werden.

Gene werden im Internet typischerweise ein Dateiformat haben, das einen riesigen String mit sämtlichen Nukleotiden in der Sequenz des Gens enthält. Das nächste Listing zeigt ein Beispiel, wie ein Gen-String aussehen kann.

```
String geneStr = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

Listing 2.3 Gen-String-Beispiel

Der einzige Zustand eines `Gene` ist ein `ArrayList` aus `Codons`. Außerdem brauchen wir einen Konstruktor, der einen Gen-String in ein Gen umwandelt (einen `String` in ein `ArrayList` aus `Codons`).

```
private ArrayList<Codon> codons = new ArrayList<>();

public Gene(String geneStr) {
    for (int i = 0; i < geneStr.length() - 3; i += 3) {
```

```
        // Alle 3 Zeichen im String nehmen und ein Codon formen
        codons.add(new Codon(geneStr.substring(i, i + 3)));
    }
}
```

Listing 2.4 Gene.java (Fortsetzung)

Dieser Konstruktor durchwandert kontinuierlich den übergebenen `String` und konvertiert seine jeweils nächsten drei Zeichen in `Codon`s, die am Ende eines neuen `Gene`-Objekts hinzugefügt werden. Er setzt auf den Konstruktor von `Codon` auf, der die Umwandlung eines `String` aus drei Buchstaben in ein `Codon` beherrscht.

2.1.2 Lineare Suche

Eine grundlegende Operation, die wir auf einem Gen ausführen wollen, besteht darin, nach einem bestimmten Codon zu suchen. Ein Wissenschaftler würde auf diese Weise vielleicht sehen wollen, ob es für eine bestimmte Aminosäure kodiert. Das Ziel ist, einfach herauszufinden, ob das Codon innerhalb des Gens existiert oder nicht.

Eine lineare Suche durchwandert jedes Element in einem Suchraum in der Reihenfolge der ursprünglichen Datenstruktur, bis das Gesuchte gefunden oder das Ende der Datenstruktur erreicht wird. Im Grunde ist eine lineare Suche die einfachste, natürlichste und offensichtlichste Art, nach etwas zu suchen. Im ungünstigsten Fall muss eine lineare Suche jedes Element einer Datenstruktur betrachten, sodass es von der Komplexität O(*n*) ist, wobei *n* die Anzahl der Elemente in der Struktur ist. Dies wird in Abbildung 2.2 veranschaulicht.

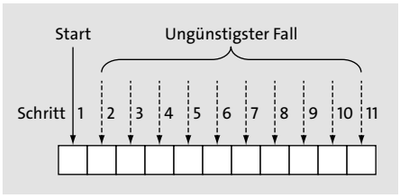

Abbildung 2.2 Im ungünstigsten Fall einer linearen Suche müssen Sie nacheinander jedes Element des Arrays betrachten.

Es ist trivial, eine Funktion zu definieren, die eine lineare Suche durchführt. Sie muss einfach jedes Element in einer Datenstruktur durchgehen und auf Gleichheit mit dem gesuchten Element überprüfen. Sie können es mit dem folgenden Code in `main()` ausprobieren.


```
public boolean linearContains(Codon key) {
    for (Codon codon : codons) {
        if (codon.compareTo(key) == 0) {
            return true; // Übereinstimmung gefunden
        }
    }
    return false;
}

public static void main(String[] args) {
    String geneStr =
 "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT";
    Gene myGene = new Gene(geneStr);
    Codon acg = new Codon("ACG");
    Codon gat = new Codon("GAT");
    System.out.println(myGene.linearContains(acg)); // true
    System.out.println(myGene.linearContains(gat)); // false
}
```

Listing 2.5 Gene.java (Fortsetzung)

Hinweis

Diese Funktion dient lediglich illustrativen Zwecken. Alle Klassen in der Java-Standardbibliothek, die die Schnittstelle `Collection` implementieren (wie etwa `ArrayList` und `LinkedList`) verfügen über eine `contains()`-Methode, die wahrscheinlich besser optimiert ist, als alles, was wir selbst schreiben würden.

2.1.3 Binärsuche

Es gibt eine schnellere Methode, als jedes Element zu betrachten, aber für diese müssen wir vorab über die Sortierreihenfolge der Datenstruktur Bescheid wissen. Wenn wir wissen, dass die Struktur sortiert ist, und auf jedes Element darin unmittelbar über seinen Index zugreifen können, können wir eine Binärsuche durchführen.

Eine Binärsuche funktioniert so: Sie betrachtet das mittlere Element einer sortierten Abfolge von Elementen, vergleicht es mit dem gesuchten Element, verkleinert den Suchbereich aufgrund dieses Vergleichs und startet den Prozess erneut. Schauen wir uns ein konkretes Beispiel an.

Angenommen, wir haben eine Liste alphabetisch sortierter Wörter wie `["Hund", "Känguru", "Katze", "Lama", "Marder", "Ratte", "Zebra"]` und suchen nach dem Wort »Ratte«:

1. Wir könnten feststellen, dass das mittlere Element in dieser Liste aus sieben Wörtern »Lama« ist.
2. Wir könnten feststellen, dass »Ratte« im Alphabet nach »Lama« kommt, also muss es sich (näherungsweise) in der Hälfte der Liste befinden, die nach »Lama« kommt. (Hätten wir »Ratte« in diesem Schritt gefunden, könnten wir den Fundort zurückgeben; hätte es sich herausgestellt, dass unser Wort vor dem überprüften mittleren Wort kommt, könnten wir sicher sein, dass es sich in der Hälfte der Liste vor »Lama« befindet.)
3. Wir könnten die Schritte 1 und 2 für die Hälfte der Liste wiederholen, von der wir wissen, dass »Ratte« sich immer noch darin befinden kann. Im Prinzip wird diese Hälfte unsere neue Basisliste. Diese Schritte werden wiederholt ausgeführt, bis »Ratte« gefunden wird oder bis der Suchbereich keine zu durchsuchenden Elemente mehr enthält, was bedeutet, dass »Ratte« nicht in der Wortliste vorkommt.

Abbildung 2.3 veranschaulicht eine Binärsuche. Beachten Sie, dass anders als bei der linearen Suche nicht jedes Element durchsucht wird.

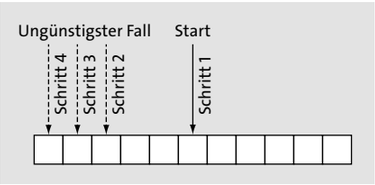

Abbildung 2.3 Im ungünstigsten Fall einer Binärsuche durchsuchen Sie nur lg(n) Elemente der Liste.

Eine Binärsuche halbiert den Suchraum immer wieder, sodass seine Laufzeit im ungünstigsten Fall O(lg *n*) beträgt. Die Sache hat jedoch einen Haken. Im Gegensatz zur linearen Suche benötigt eine binäre Suche eine sortierte Datenstruktur zum Durchsuchen, und das Sortieren benötigt Zeit. Tatsächlich braucht Sortieren mit den besten Sortieralgorithmen eine Zeit von O(*n* lg *n*). Wenn wir unsere Suche also nur einmal durchführen und unsere ursprüngliche Datenstruktur unsortiert ist, ist es wahrscheinlich sinnvoller, einfach eine lineare Suche durchzuführen. Aber wenn die Suche viele Male durchgeführt wird, lohnt sich der Zeitaufwand für das Sortieren, da der Nutzen der stark reduzierten Dauer jeder einzelnen Suche überwiegt.

Eine Binärsuchfunktion für ein Gen und ein Codon unterscheidet sich nicht von einer für jede andere Art von Daten, weil der Typ `Codon` anderen Typen ähnelt und der Typ `Gene` lediglich eine `ArrayList` aus `Codons` enthält.

Beachten Sie, dass wir die Codons im folgenden Beispiel zunächst sortieren – dies hebt alle Vorteile einer binären Suche auf, da die Sortierung mehr Zeit in Anspruch nimmt als die Suche, so wie im vorherigen Abschnitt beschrieben. Zur Veranschaulichung ist die Sortierung jedoch notwendig, da wir beim Ausführen dieses Beispiels nicht wissen können, dass die `ArrayList` aus `codons` sortiert ist.

```
public boolean binaryContains(Codon key) {
    // Binäre Suche funktioniert nur mit sortierten Collections
    ArrayList<Codon> sortedCodons = new ArrayList<>(codons);
    Collections.sort(sortedCodons);
    int low = 0;
    int high = sortedCodons.size() - 1;
    while (low <= high) { // Solange es noch einen Suchraum gibt
        int middle = (low + high) / 2;
        int comparison = codons.get(middle).compareTo(key);
        if (comparison < 0) { // Mittleres Codon < key
            low = middle + 1;
        } else if (comparison > 0) { // Mittleres Codon > key
            high = middle - 1;
        } else { // Mittleres Codon gleich key
            return true;
        }
    }
    return false;
}
```

Listing 2.6 Gene.java (Fortsetzung)

Schauen wir uns diese Funktion Zeile für Zeile an.

```
int low = 0;
int high = sortedCodons.size() - 1;
```

Wir beginnen mit einem Suchbereich, der die gesamte Liste (das Gen) umfasst.

```
while (low <= high) {
```

Wir suchen weiter, solange es noch einen Bereich gibt, in dem gesucht werden kann. Wenn `low` größer als `high` ist, bedeutet dies, dass es in der Liste keine Einträge mehr gibt, die wir uns anschauen könnten.

```
int middle = (low + high) / 2;
```

Wir berechnen die Mitte `middle`, indem wir Integer-Division und die einfache Mittelwert-Formel verwenden, die Sie in der Schule gelernt haben.

```
int comparison = codons.get(middle).compareTo(key);
if (comparison < 0) { // Mittleres Codon < key
    low = middle + 1;
```

Wenn das Element, das wir suchen, nach dem mittleren Element des betrachteten Bereichs kommt, modifizieren wir den Bereich, den wir uns im nächsten Durchlauf der Schleife anschauen, indem wir `low` auf die Position gleich hinter dem aktuellen mittleren Element verschieben. Auf diese Weise haben wir den Bereich für die nächste Iteration halbiert.

```
} else if (comparison > 0) { // Mittleres Codon > key
    high = middle - 1;
```

Entsprechend halbieren wir in die andere Richtung, wenn das gesuchte Element kleiner als das mittlere Element ist.

```
} else { // Mittleres Codon gleich key
    return true;
}
```

Wenn das gesuchte Element weder kleiner noch größer als das mittlere Element ist, heißt das, dass wir es gefunden haben! Und natürlich geben wir `false` zurück (hier nicht nochmals abgedruckt), wenn es keine weiteren Schleifendurchläufe gibt, um anzuzeigen, dass es nicht gefunden wurde.

Wir können nun versuchen, unsere binäre Suchmethode mit demselben Gen und demselben Codon auszuführen. Wir können `main()` probehalber verändern.

```
public static void main(String[] args) {
    String geneStr =
     "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT";
    Gene myGene = new Gene(geneStr);
    Codon acg = new Codon("ACG");
```



```
    Codon gat = new Codon("GAT");
    System.out.println(myGene.linearContains(acg)); // true
    System.out.println(myGene.linearContains(gat)); // false
    System.out.println(myGene.binaryContains(acg)); // true
    System.out.println(myGene.binaryContains(gat)); // false
}
```

Listing 2.7 Gene.java (Fortsetzung)

> **Tipp**
>
> Genau wie bei der linearen Suche müssen Sie auch die binäre Suche nicht selbst implementieren, da es eine Implementierung in der Java-Standardbibliothek gibt. Die Methode `Collections.binarySearch()` kann jede sortierte `Collection` (wie etwa eine sortierte `ArrayList`) durchsuchen.

2.1.4 Ein generisches Beispiel

Die Methoden `linearContains()` und `binaryContains()`lassen sich so verallgemeinern, dass sie mit so gut wie jeder Java-`List` arbeiten können. Die folgenden verallgemeinerten Versionen sind beinahe identisch mit denjenigen, die Sie bereits gesehen haben, nur einige Namen und Typen wurden ausgetauscht.

> **Hinweis**
>
> Das nachfolgende Codelisting enthält viele importierte Typen. Wir werden die Datei *GenericSearch.java* für viele weitere generische Suchalgorithmen in diesem Kapitel verwenden und haben die dafür notwendigen Importe so bereits abgehandelt.

> **Hinweis**
>
> Das Keyword `extends` in `T extends Comparable<T>` bedeutet, dass `T` einen Typ haben muss, der das Interface `Comparable` implementiert.

```
package chapter2;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
```

```
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.ToDoubleFunction;

public class GenericSearch {

    public static <T extends Comparable<T>> boolean linearContains(
     List<T> list, T key) {
        for (T item : list) {
            if (item.compareTo(key) == 0) {
                return true; // Übereinstimmung gefunden
            }
        }
        return false;
    }

    // *list* bereits sortiert
    public static <T extends Comparable<T>> boolean binaryContains(
     List<T> list, T key) {
        int low = 0;
        int high = list.size() - 1;
        while (low <= high) {
            int middle = (low + high) / 2;
            int comparison = list.get(middle).compareTo(key);
            if (comparison < 0) { // Mittleres Codon < key
                low = middle + 1;
            } else if (comparison > 0) { // Mittleres Codon > key
                high = middle - 1;
            } else { // Mittleres Codon gleich key
                return true;
            }
        }
        return false;
    }
```



```
    public static void main(String[] args) {
        System.out.println(linearContains(
         List.of(1, 5, 15, 15, 15, 15, 20), 5)); // true
        System.out.println(binaryContains(
         List.of("a", "d", "e", "f", "z"), "f")); // true
        System.out.println(binaryContains(
         List.of("john", "mark", "ronald", "sarah"), "sheila")); // false
    }

}
```

Listing 2.8 GenericSearch.java

Nun können Sie versuchen, nach anderen Datentypen zu suchen. Diese Methoden funktionieren mit jeder Liste aus Vergleichswerten. Darin liegt die Macht generisch geschriebenen Codes.

2.2 Labyrinthe lösen

Einen Pfad durch ein Labyrinth zu finden, ist eine Analogie für viele gängige Suchaufgaben in der Informatik. Warum dann also nicht wortwörtlich einen Pfad durch ein Labyrinth finden, um Breitensuche-, Tiefensuche- und A*-Algorithmen zu veranschaulichen?

Unser Labyrinth sei ein zweidimensionales Gitter aus `Cell`-Objekten. Eine `Cell` ist ein `enum`, das weiß, wie es sich selbst in einen String verwandeln kann. Zum Beispiel stellt `" "` einen leeren Platz und `"X"` einen besetzten Platz dar. Es gibt noch weitere Fälle, die bei der Ausgabe eines Labyrinths zu Darstellungszwecken verwendet werden.

```
package chapter2;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import chapter2.GenericSearch.Node;

public class Maze {

    public enum Cell {
        EMPTY(" "),
        BLOCKED("X"),
```

Kapitel 3
Bedingungserfüllungsprobleme

Viele Aufgaben, für deren Lösung Computer eingesetzt werden, lassen sich im weitesten Sinn als Bedingungserfüllungsprobleme (Constraint-Satisfaction-Probleme, CSPs) kategorisieren. CSPs bestehen aus *Variablen* mit möglichen Werten, deren mögliche Werte in Bereiche fallen, die als *Domänen* (Wertebereiche) bezeichnet werden. *Bedingungen* (Constraints) zwischen den Variablen müssen erfüllt werden, damit Bedingungserfüllungsprobleme gelöst werden. Diese drei Konzepte – Variablen, Domänen und Bedingungen – sind einfach zu verstehen, und auf ihrer Allgemeingültigkeit basiert die große Anzahl von Anwendungsgebieten für die Lösung von Bedingungserfüllungsproblemen.

Betrachten wir eine Beispielaufgabe. Angenommen, Sie möchten ein Freitagsmeeting für Joe, Mary und Sue planen. Sue muss mit mindestens einer weiteren Person an dem Meeting teilnehmen. Für diese Planungsaufgabe seien die drei Personen – Joe, Mary und Sue – die Variablen. Die Domäne für jede Variable seien ihre jeweiligen Verfügbarkeitsstunden.

Beispielsweise hat die Variable Mary die Domäne 14 Uhr, 15 Uhr und 16 Uhr. Dieses Problem hat auch zwei Bedingungen. Die erste ist, dass Sue an dem Meeting teilnehmen muss. Die andere ist, dass mindestens zwei Personen an dem Meeting teilnehmen müssen. Einem Lösungsalgorithmus für Bedingungserfüllungsprobleme werden die drei Variablen, drei Domänen und zwei Bedingungen übergeben, und er wird das Problem lösen, ohne dass der Benutzer genau erklären muss, *wie*. Abbildung 3.1 veranschaulicht das Beispiel.

Programmiersprachen wie Prolog und Picat enthalten eingebaute Elemente zum Lösen von Bedingungserfüllungsproblemen. Das übliche Verfahren in anderen Sprachen besteht in der Erstellung eines Frameworks, das eine Backtracking-Suche und diverse Heuristiken zur Performanceverbesserung dieser Suche enthält. In diesem Kapitel erstellen wir zuerst ein Framework für CSPs, das diese mithilfe einer einfachen rekursiven Backtracking-Suche löst. Dann verwenden wir das Framework, um mehrere unterschiedliche Beispielaufgaben zu lösen.

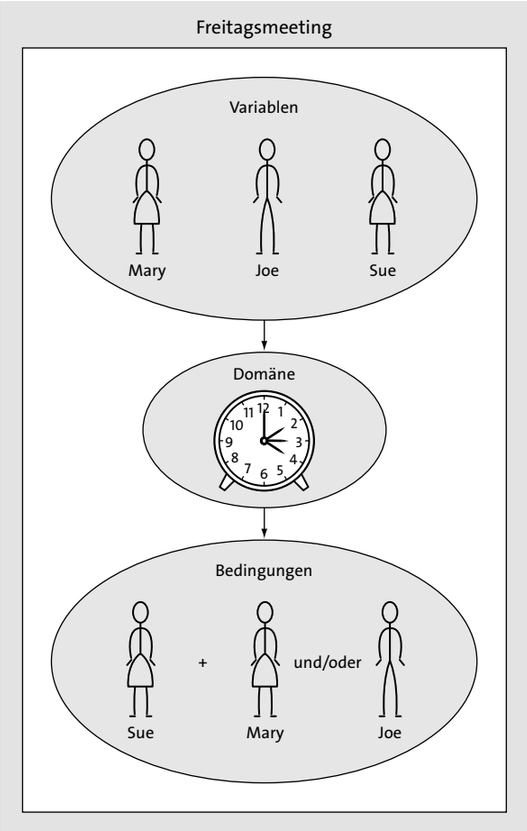

Abbildung 3.1 Terminplanungsaufgaben sind eine klassische Anwendung von Bedingungserfüllung -Frameworks.

3.1 Ein Framework für Bedingungserfüllungsprobleme schreiben

Bedingungen werden als Unterklassen einer `Constraint`-Klasse definiert. Jede `Constraint` besteht aus den `variables`, für die sie Bedingungen setzt, und der Methode `satisfied()`, die überprüft, ob sie erfüllt sind. Zu bestimmen, ob eine Bedingung erfüllt ist, ist die Hauptlogik hinter der Definition eines bestimmten Bedingungserfüllungsproblems.

Die Standardimplementierung sollte überschrieben werden. Das muss sogar geschehen, weil wir unsere `Constraint`-Klasse als abstrakte Basisklasse definieren. Abstrakte Basisklassen sind nicht dafür gedacht, instanziiert zu werden. Stattdessen sind nur ihre Unterklassen, die ihre `abstract`-Methoden überschreiben und implementieren, von praktischem Nutzen.


```
package chapter3;

import java.util.List;
import java.util.Map;
// V ist der Variablentyp, D ist der Domänentyp
public abstract class Constraint<V, D> {

    // Die Variablen, zwischen denen die Bedingung besteht
    protected List<V> variables;

    public Constraint(List<V> variables) {
        this.variables = variables;
    }

    // Muss von Unterklassen überschrieben werden
    public abstract boolean satisfied(Map<V, D> assignment);
}
```

Listing 3.1 Constraint.java

> **Tipp**
>
> In Java kann die Entscheidung zwischen einer abstrakten Klasse und einer Schnittstelle schwierig sein. Nur abstrakte Klassen können Instanzvariablen haben. Da wir die Instanzvariable `variables` verwenden, entscheiden wir uns hier für eine abstrakte Klasse.

Das Kernstück unseres Bedingungserfüllungs-Frameworks bildet eine Klasse namens `CSP`. `CSP` ist der Sammelpunkt für Variablen, Domänen und Bedingungen. Sie verwendet Generics, damit sie flexibel genug ist, um mit jeder Art von Variablen- und Domänenwerten zu arbeiten (`V` Schlüssel und `D` Domänenwerte). Innerhalb von `CSP` haben die Collections `variables`, `domains` und `constraints` Typen, die Sie erwarten würden. Die Collection `variables` ist eine `List` von Variablen, `domains` ist eine `Map`, die Variablen den Listen möglicher Werte zuordnet (die Domänen dieser Variablen), und `constraints` ist eine `Map`, die jede Variable einer `List` der für sie geltenden Bedingungen zuordnet.

```
package chapter3;

import java.util.ArrayList;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CSP<V, D> {
    private List<V> variables;
    private Map<V, List<D>> domains;
    private Map<V, List<Constraint<V, D>>> constraints = new HashMap<>();

    public CSP(List<V> variables, Map<V, List<D>> domains) {
        this.variables = variables;
        this.domains = domains;
        for (V variable : variables) {
            constraints.put(variable, new ArrayList<>());
            if (!domains.containsKey(variable)) {
                throw new IllegalArgumentException(
     "Jeder Variablen sollte eine Domäne zugewiesen sein.");
            }
        }
    }
    public void addConstraint(Constraint<V, D> constraint) {
        for (V variable : constraint.variables) {
            if (!variables.contains(variable)) {
                throw new IllegalArgumentException(
     "Variable in Bedingung nicht in CSP");
            }
            constraints.get(variable).add(constraint);
        }
    }
```

Listing 3.2 CSP.java

Der Konstruktor erzeugt die `Map constraints`. Die Methode `Constraint()` durchläuft alle Variablen, die von der jeweiligen Bedingung betroffen sind, und fügt sich zur `constraints`-Zuordnung von jeder von ihnen hinzu. Beide Methoden enthalten einfache Fehlerprüfungen und lösen eine Exception aus, wenn eine `variable` keine Domäne hat oder wenn eine `constraint` auf eine nicht existierende Variable gesetzt wird.

Woher wissen wir, ob eine gegebene Konfiguration von Variablen und ausgewählten Domänenwerten die Bedingungen erfüllt? Wir nennen eine solche gegebene Konfiguration eine *Zuordnung*. Mit anderen Worten: Eine Zuordnung ist ein bestimmter Domänenwert, der für jede Variable ausgewählt wird. Wir brauchen eine Funktion, die jede Bedingung für eine gegebene Variable mit einer Zuordnung vergleicht, um zu sehen, ob der Variablenwert in der Zuordnung die Bedingungen erfüllt. Hier implementieren wir die Funktion `consistent()` als Methode von `CSP`.


```
    // Herausfinden, ob die Wertzuordnung konsistent ist, indem alle
    // Bedingungen für die angegebene Variable darauf geprüft werden
    public boolean consistent(V variable, Map<V, D> assignment) {
        for (Constraint<V, D> constraint : constraints.get(variable)) {
            if (!constraint.satisfied(assignment)) {
                return false;
            }
        }
        return true;
    }
```

Listing 3.3 CSP.java (Fortsetzung)

`consistent()` durchläuft jede Bedingung für eine gegebene Variable (immer diejenige Variable, die gerade zur Zuordnung hinzugefügt wurde) und überprüft, ob die Bedingung unter Berücksichtigung der neuen Zuordnung erfüllt ist. Wenn die Zuordnung jede Bedingung erfüllt, wird `true` zurückgegeben. Wenn eine der Variablen auferlegte Bedingung nicht erfüllt ist, wird `false` zurückgegeben.

Dieses Bedingungserfüllungs-Framework verwendet eine einfache Backtracking-Suche, um Lösungen für Probleme zu finden. *Backtracking* ist ein Verfahren, bei dem Sie jedes Mal, wenn Ihre Suche auf eine Wand stößt, zum letzten Entscheidungspunkt vor der Wand zurückgehen und einen anderen Pfad wählen. Wenn Sie finden, dass sich das sehr wie die Tiefensuche aus Kapitel 2, »Suchaufgaben«, anhört, haben Sie gut aufgepasst. Die Backtracking-Suche, die in der folgenden Methode `backtrackingSearch()` implementiert wird, ist eine Art rekursiver Tiefensuche, die Ideen kombiniert, die Sie in Kapitel 1, »Kleine Aufgaben«, und Kapitel 2, »Suchaufgaben«, kennengelernt haben. Wir implementieren außerdem eine Hilfsmethode, die einfach nur `backtrackingSearch()` mit einer leeren initialen `Map` aufruft. Die Hilfsmethode ist nützlich, um eine Suche anzustoßen.

```
    public Map<V, D> backtrackingSearch(Map<V, D> assignment) {
  // Zuordnung vollständig, wenn jede Variable zugeordnet ist (Abbruchbedingung)
        if (assignment.size() == variables.size()) {
            return assignment;
        }
```

```
        // Erste Variable im CSP, aber nicht in der Zuordnung
        V unassigned = variables.stream().filter(
         v -> !assignment.containsKey(v)).findFirst().get();
        // Alle Domänenwerte der ersten nicht zugewiesenen Variablen
        // durchgehen
        for (D value : domains.get(unassigned)) {
            // Flache Kopie der Zuordnung, die wir verändern können
            Map<V, D> localAssignment = new HashMap<>(assignment);
            localAssignment.put(unassigned, value);
            // Wenn wir noch konsistent sind, Rekursion (fortfahren)
            if (consistent(unassigned, localAssignment)) {
                Map<V, D> result = backtrackingSearch(localAssignment);
                // Wenn wir das Ergebnis nicht gefunden haben, Backtracking
                if (result != null) {
                    return result;
                }
            }
        }
        return null;
    }

    // Hilfsfunktion für backtrackingSearch, wenn noch nichts bekannt ist
    public Map<V, D> backtrackingSearch() {
        return backtrackingSearch(new HashMap<>());
    }
}
```

Listing 3.4 CSP.java (Fortsetzung)

Schauen wir uns `backtrackingSearch()` Zeile für Zeile an.

```
if (assignment.size() == variables.size()) {
    return assignment;
}
```

Die Abbruchbedingung für die rekursive Suche besteht darin, eine gültige Zuordnung für jede Variable gefunden zu haben. Wenn dies der Fall ist, geben wir die erste Instanz einer gültigen Lösung zurück. (Wir suchen nicht weiter.)

```
V unassigned = variables.stream().filter(
 v -> !assignment.containsKey(v)).findFirst().get();
```


Um eine neue Variable auszuwählen, deren Domäne wir untersuchen, gehen wir einfach alle Variablen durch und finden die erste, die noch keine Zuordnung hat. Dafür erstellen wir einen `Stream` von `variables`, der danach gefiltert wird, ob sie zugewiesen sind, und wir ziehen uns die erste, die nicht zugewiesen ist, mit `findFirst()`. `filter()` erwartet ein `Predicate`. Ein solches Prädikat ist ein funktionales Interface, das eine Funktion beschreibt, die ein Argument annimmt und ein `boolean` zurückliefert. Unser Prädikat ist ein Lambdaausdruck (`v -> !assignment.contains-Key(v)`), der `true` zurückliefert, wenn `assignment` das Argument nicht enthält, welches in unserem Fall eine Variable für unser `CSP` ist.

```
for (D value : domains.get(unassigned)) {
    Map<V, D> localAssignment = new HashMap<>(assignment);
    localAssignment.put(unassigned, value);
```

Wir versuchen, der Variablen nacheinander alle möglichen Domänenwerte zuzuweisen. Die neue Zuordnung für jeden Wert wird in einer lokalen Map namens `localAssignment` gespeichert.

```
if (consistent(unassigned, localAssignment)) {
    Map<V, D> result = backtrackingSearch(localAssignment);
    if (result != null) {
        return result;
    }
}
```

Wenn die neue Zuordnung in `localAssignment` mit allen Bedingungen konsistent ist (der Fall, auf den `consistent()` prüft), fahren wir unter Beibehaltung der neuen Zuordnung mit der rekursiven Suche fort. Wenn die neue Zuordnung sich als vollständig herausstellt (die Abbruchbedingung), geben wir die neue Zuordnung entlang der Rekursionskette zurück.

```
return null;
```

Wenn wir schließlich jeden möglichen Domänenwert für eine bestimmte Variable durchgegangen sind und es keine Lösung unter Verwendung des bestehenden Satzes von Zuordnungen gibt, geben wir `null` zurück, was bedeutet, dass es keine Lösung gibt. Dies führt zum Backtracking die Rekursionskette hinauf bis zu dem Punkt, wo eine andere Zuordnung hätte gemacht werden können.

3.2 Die Landkarte Australiens einfärben

Stellen Sie sich vor, Sie haben eine Landkarte von Australien, die Sie nach Staat/Territorium (hier der Einfachheit halber »Regionen« genannt) einfärben möchten. Keine zwei aneinandergrenzenden Regionen sollen dieselbe Farbe haben. Können Sie die Regionen mit nur drei unterschiedlichen Farben einfärben?

Die Antwort ist ja. Probieren Sie es selbst aus. (Das geht am einfachsten, wenn Sie eine Karte von Australien mit weißem Hintergrund ausdrucken.) Als Menschen können wir durch Überlegen und etwas Ausprobieren schnell die Lösung herausfinden. Es ist eine wirklich triviale Aufgabe und eine großartige erste Aufgabe für unser Bedingungserfüllungs-Lösungsverfahren mit Backtracking. Eine Lösung der Aufgabe wird in Abbildung 3.2 veranschaulicht.

Abbildung 3.2 In einer Lösung zur Aufgabe, die Landkarte Australiens einzufärben, dürfen keine zwei Teile Australiens, die aneinander angrenzen, dieselbe Farbe haben.

Um die Aufgabe als CSP zu modellieren, definieren wir die Variablen, Domänen und Bedingungen. Die Variablen sind die sieben Regionen von Australien (zumindest die sieben, auf die wir uns beschränken): Western Australia, Northern Territory, South Australia, Queensland, New South Wales, Victoria und Tasmania. In unserem CSP können sie durch Strings modelliert werden. Die Domäne jeder Variablen sind die drei Farben, die zugewiesen werden können. (Wir verwenden Rot, Grün und Blau.) Die Bedingungen sind der schwierige Teil. Keine zwei aneinandergrenzenden Regionen dürfen mit derselben Farbe eingefärbt werden, sodass unsere Bedingungen darauf basieren, welche Regionen aneinandergrenzen. Dafür können wir sogenannte *Binärbedingungen* (Bedingungen zwischen zwei Variablen) verwenden. Je zwei Regionen, die eine gemeinsame Grenze haben, haben auch eine gemeinsame Binärbedingung, die anzeigt, dass ihnen nicht dieselbe Farbe zugewiesen werden darf.

Um diese binären Bedingungen in Code zu implementieren, müssen wir eine Unterklasse der Klasse `Constraint` erstellen. Die Unterklasse `MapColoringConstraint` nimmt in ihrem Konstruktor zwei Variablen entgegen: die beiden Regionen, die eine gemeinsame Grenze haben. Die überschriebene Methode `satisfied()` überprüft zunächst, ob den beiden Regionen bereits Domänenwerte (Farben) zugewiesen wurden; wenn eine noch keine hat, ist die Bedingung trivial erfüllt, bis dies der Fall ist. (Es kann keinen Konflikt geben, wenn eine von ihnen noch keine Farbe hat.) Dann prüft die Methode, ob beiden Regionen dieselbe Farbe zugeordnet wurde. Dies ist offensichtlich ein Konflikt, die Bedingung ist also nicht erfüllt, wenn die Farben identisch sind.

Die Klasse wird hier bis auf `main()` vollständig gezeigt. `MapColoring-Constraint` selbst ist nicht generisch, sondern leitet eine Version von der generischen Klasse `Constraint` ab, deren Parameter anzeigen, dass sowohl Variablen als auch Domänen vom Typ `String` sind.

```
package chapter3;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public final class MapColoringConstraint extends Constraint<String, String> {
    private String place1, place2;

    public MapColoringConstraint(String place1, String place2) {
        super(List.of(place1, place2));
        this.place1 = place1;
        this.place2 = place2;
```

```
    }

    @Override
    public boolean satisfied(Map<String, String> assignment) {
        // Wenn einer der beiden Orte noch nicht in der Zuordnung ist, dann
        // ist es noch nicht möglich, dass ihre Farben in Konflikt geraten
        if (!assignment.containsKey(place1) ||
            !assignment.containsKey(place2)) {
            return true;
        }
        // Prüfen, ob die place1 zugeordnete Farbe nicht dieselbe ist
        // wie die place2 zugeordnete Farbe
        return !assignment.get(place1).equals(assignment.get(place2));
    }
```

Listing 3.5 MapColoringConstraint.java

Da wir nun eine Möglichkeit haben, die Bedingungen zwischen Regionen zu implementieren, ist die Ausarbeitung der Einfärbeaufgabe der Landkarte Australiens mit unserem CSP-Löser nur noch eine Frage des Ausfüllens der Domänen und Variablen und des anschließenden Hinzufügens von Bedingungen.

```
    public static void main(String[] args) {
        List<String> variables =
     List.of("Western Australia", "Northern Territory", "South Australia",
     "Queensland", "New South Wales", "Victoria", "Tasmania");
        Map<String, List<String>> domains = new HashMap<>();
        for (String variable : variables) {
            domains.put(variable, List.of("rot", "grün", "blau"));
        }
        CSP<String, String> csp = new CSP<>(variables, domains);
        csp.addConstraint(new MapColoringConstraint("Western Australia",
     "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Western Australia",
     "South Australia"));
        csp.addConstraint(new MapColoringConstraint("South Australia",
     "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
     "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
     "South Australia"));
```



```
        csp.addConstraint(new MapColoringConstraint("Queensland",
    "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("New South Wales",
    "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
    "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
    "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("Victoria", "Tasmania"));
```

Listing 3.6 MapColoringConstraint.java (Fortsetzung)

Schließlich wird `backtrackingSearch()` aufgerufen, um eine Lösung zu finden.

```
        Map<String, String> solution = csp.backtrackingSearch();
        if (solution == null) {
            System.out.println("Keine Lösung gefunden!");
        } else {
            System.out.println(solution);
        }
    }

}
```

Listing 3.7 MapColoringConstraint.java (Fortsetzung)

Eine korrekte Lösung wird eine Farbzuweisung für jede Region enthalten.

```
{Western Australia=rot, New South Wales=grün, Victoria=rot, Tasmania=
grün, Northern Territory=grün, South Australia=blau, Queensland=rot}
```

3.3 Das Acht-Damen-Problem

Ein Schachbrett ist ein Raster von acht mal acht Quadraten. Eine Dame ist eine Schachfigur, die sich auf dem Schachbrett beliebig viele Felder weit waagerecht, senkrecht oder diagonal bewegen kann. Eine Dame bedroht eine andere Figur, wenn sie das Feld dieser Figur in einem Zug erreichen kann, ohne eine andere Figur zu überspringen. (Mit anderen Worten wird jede Figur, die sich in der Sichtlinie der Dame befindet, von dieser bedroht.) Das Acht-Damen-Problem stellt die Frage, wie acht Damen auf dem Schachbrett platziert werden können, ohne dass irgendeine Dame irgendeine andere bedroht. Eine von vielen möglichen Lösungen der Aufgabe wird in Abbildung 3.3 veranschaulicht.