Kapitel 5
Spiele für eine Person lösen

1996 schreibt der IBM-Rechner Deep Blue Geschichte und besiegt den Schachweltmeister Gari Kasparow. Die Niederlage des höchstrangigen Go-Spielers Lee Sedol gegen das Programm AlphaGo hat ebenfalls weltweites Aufsehen erregt. Die Entwicklung von Computerprogrammen, die menschlichen Spielern bei Schach & Co. überlegen sind, ist seit den frühesten Tagen der KI ein zentrales Forschungs- und Experimentierfeld. Wir stellen grundlegende Techniken der Spiel-KI vor.



Worum es in diesem Kapitel geht

- Das Spiel »Fruchtkräsch«, das von Puzzlespielen wie Candy Crush oder Collapse inspiriert ist.
- Eine KI, die Fruchtkräsch und andere *zugbasierte Ein-Personen-Spiele* konkurrenzlos gut spielt.
- Eine Funktion, die Spielstände bewertet.
- Eine *Baumstruktur*, die alle denkbaren Spielverläufe darstellt.
- Die Durchsuchung dieses Baums per *Warteschlangenverfahren*, um den besten Zug zu finden.

In diesem und dem nächsten Kapitel geht es um Anwendungen, die zugbasierte Spiele erfolgreich spielen. Zunächst stellen wir eine auf Ein-Personen-Spiele spezialisierte KI vor. Im nächsten Kapitel wenden wir uns Spielen für zwei Personen zu.

5.1 Das Spiel Fruchtkräsch

Das Beispielprogramm »Fruchtkräsch« basiert auf einem weithin bekannten Spielprinzip: Auf einer rechteckigen Spielfläche sind verschiedenfarbige Spielsteine aufgestapelt. Zusammenhängende Gruppen von mindestens drei gleichfarbigen Steinen – hier *gleichfarbige Gruppen* genannt – können Sie per Mausklick entfernen. Für jeden entfernten Stein erhalten Sie einen Punkt. Die entstehenden Lücken werden gefüllt, indem darüberliegende Steine »herunterfallen«, also nach unten aufrücken. Auch leere Spalten werden geschlossen, indem rechts davon befindliche Spalten nach links aufrücken.

Persönliches Exemplar für Pit Noack

Das Spiel ist beendet, wenn keine gleichfarbige Gruppe mehr vorhanden und damit kein Zug mehr möglich ist. Ziel des Spiels ist es, möglichst viele Steine zu entfernen, also möglichst viele Punkte zu bekommen. Reiz und Herausforderung bestehen darin, dass durch aufrückende Steine neue gleichfarbige Gruppen entstehen können. Es macht also einen Unterschied, in welcher Reihenfolge Sie die Steine wegklicken.

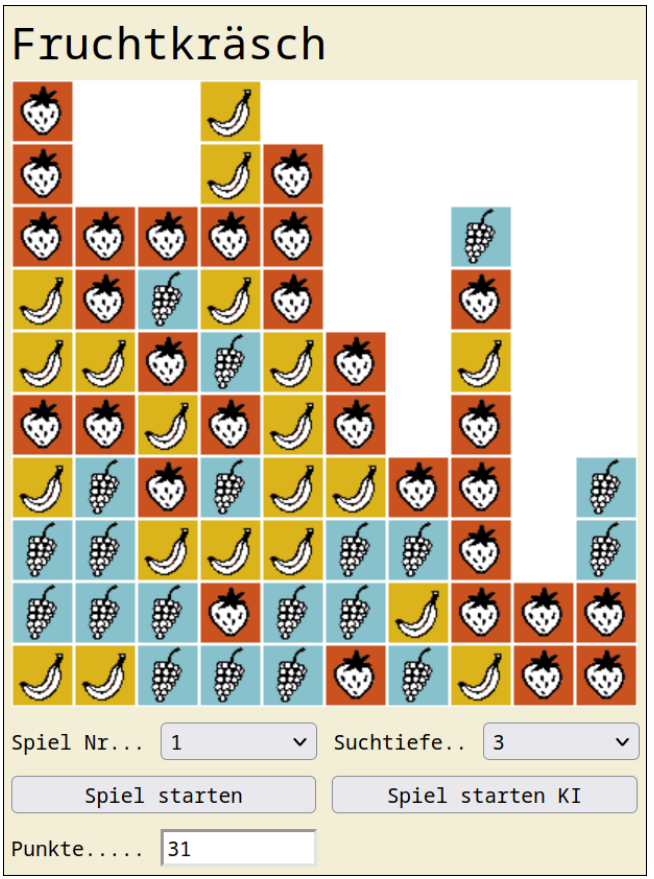


Abbildung 5.1 Das Beispielprogramm »Fruchtkräsch«

Sie können per Klick auf die Schaltfläche SPIEL STARTEN eine eigene Partie beginnen oder mit einem Klick auf SPIEL STARTEN KI dem Computer beim Spielen zuschauen. Mittels des Parameters SUCHTIEFE steuern Sie die Stärke der KI – was es damit auf sich hat, erfahren Sie weiter unten. Mittels des Drop-down-Menüs SPIEL NR. wählen Sie einen von zehn reproduzierbaren Startzuständen aus. Auf diese Weise können Sie Ihre Fruchtkräsch-Skills direkt mit jenen der KI messen oder die Spielergebnisse der KI bei unterschiedlichen Suchtiefen vergleichen. Probieren Sie es aus!

© Rheinwerk Verlag, Bonn 2023

5.2 Wie findet die KI den besten Zug?

Ob Mensch oder Maschine: Immer geht es bei *zugbasierten* Spielen wie Fruchtkräsch darum, *den jeweils besten nächsten Zug aus der Menge aller möglichen Züge zu wählen*. Aber wie entscheiden wir, welche Züge besser und welche schlechter sind? Bei Fruchtkräsch könnten wir eine Strategie wie »Spielfeld von oben nach unten abräumen« verfolgen. Oder wir stellen uns vor, welche Folgen die zur Auswahl stehenden Züge auf den weiteren Spielverlauf haben – um auf diese Weise einen vorteilhaften Zug zu identifizieren.

Auf dem zuletzt genannten Prinzip basiert auch unsere KI: Im Idealfall berechnet sie ausgehend vom aktuellen Spielzustand sämtliche denkbaren Spielverläufe bis zum Spielende. Dann *wählt die KI den Zug, der zum Spielzustand mit der besten Bewertung führt*.

Abbildung 5.2 stellt den vollständigen *Suchbaum* eines Fruchtkräsch-Spiels mit einem 5 × 5 großen Spielfeld dar. Aus dem Suchbaum können Sie alle denkbaren Spielverläufe und Endzustände ablesen, die von dem oben befindlichen *Spielzustand* aus erreichbar sind. Die Zeichnung zeigt auch, wie die KI den besten Zug ermittelt: Das ist schlicht und einfach *der erste Zug der Zugfolge, die zum Endzustand mit der höchsten Punktzahl führt*.

Im Fall eines Spielfelds mit 25 Steinen ist es ein Leichtes, alle denkbaren Spielverläufe zu berechnen. Bei größeren Spielfeldern wird das nicht funktionieren, da die Anzahl der Möglichkeiten schnell ins Astronomische wächst. Hier kommt die *Suchtiefe* ins Spiel: Sie legt fest, *wie viele Züge die KI in die Zukunft vorausberechnen soll*, und beschränkt damit auch den Rechenaufwand. Bei Suchtiefe 2 berechnet die KI sämtliche Spielzustände, die mit zwei Zügen erreichbar sind etc.

Die Begrenzung der Suchtiefe bringt ein interessantes Problem mit sich: Wir müssen jetzt auch *Spielzustände bewerten, die noch Zugmöglichkeiten bieten*. Der Punktestand allein reicht als Kriterium für die Bewertung eines Spielzustands nicht mehr aus: Angenommen, wir vergleichen zwei Spielzustände A und B, bei denen der Punktestand gleich ist – A bietet aber noch Zugmöglichkeiten, B nicht. Dann liegt es auf der Hand, dass A höher zu bewerten ist, weil durch künftige Züge noch Punkte hinzukommen werden. Unsere Bewertungsfunktion wird dies berücksichtigen.

Eine weitere Technik zur Beschränkung des Rechenaufwands besteht darin, schon vor dem Erreichen der gesetzten Suchtiefe wenig erfolgversprechende Züge auszusortieren und die daran anschließenden Spielverläufe nicht weiter zu verfolgen. Dieses Vorgehen wird als *Pruning* bezeichnet, zu Deutsch »Beschneiden«. Auch das haben wir in unsere KI eingebaut.

Persönliches Exemplar für Pit Noack

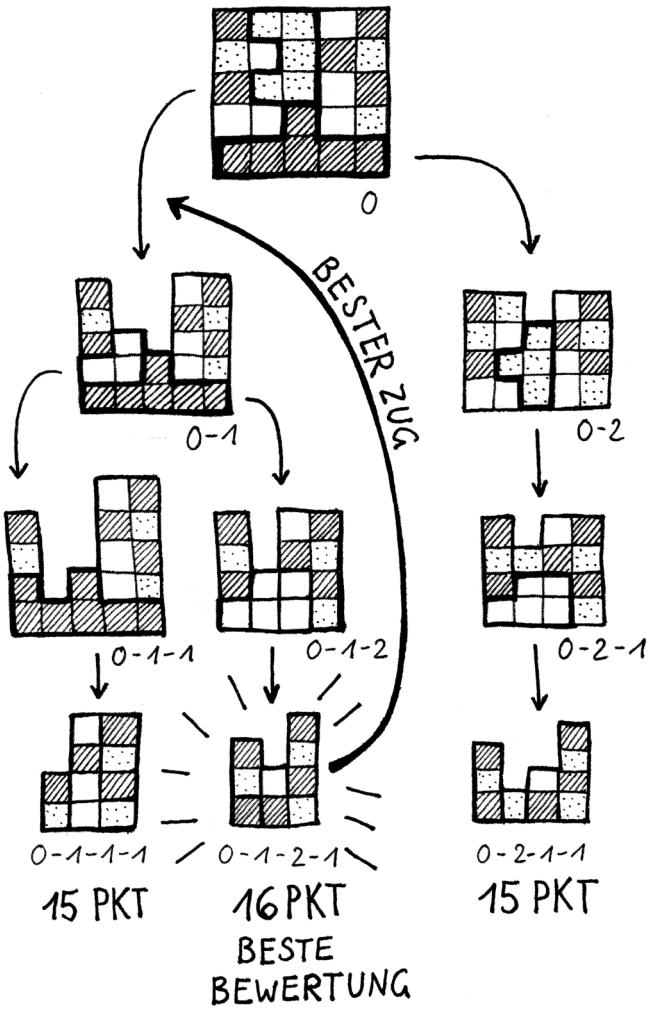


Abbildung 5.2 Die Ermittlung des besten ersten Zugs mittels einer Baumstruktur, die alle denkbaren Spielverläufe enthält

Hintergrund: Graphentheorie Teil III

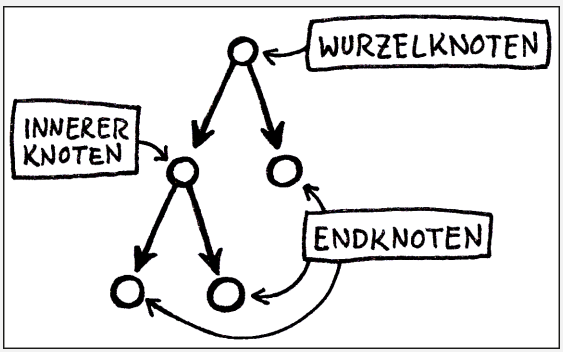
Die Baumstruktur, mit der wir in Abbildung 5.2 die möglichen Spielverläufe von »Fruchtkräsch« darstellen, kommt Ihnen vielleicht bekannt vor. Auch hier haben wir es mit einem *Graphen* zu tun: Die *Knoten* stehen für Spielzustände, die *Kanten* für gültige Züge.

© Rheinwerk Verlag, Bonn 2023

Dabei haben wir es mit einer *speziellen Klasse von Graphen* zu tun, den *gerichteten Wurzelbäumen*. Vertreter dieser Spezies zeichnen sich vereinfacht gesagt durch drei Eigenschaften aus:

- Sie besitzen einen *Wurzelknoten*, von dem aus alle weiteren Knoten erreichbar sind.
- Es führt jeweils exakt ein Weg vom Wurzelknoten zu jedem weiteren Knoten.
- Es gibt keine Rundwege.

In einem gerichteten Wurzelbaum können wir drei wesentliche Knotenarten unterscheiden: den Wurzelknoten sowie *innere Knoten* und *Endknoten*. Letztere werden auch *Blätter* genannt.

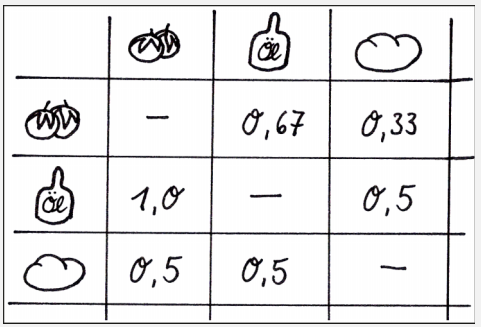


Für unsere Zwecke sind zwei weitere Begriffe nützlich: Die *Folgeknoten* eines Knotens sind sämtliche darunterliegenden Knoten, die von diesem aus erreichbar sind. Die *direkten Folgeknoten* sind dann jene, die mit einem einzigen Schritt erreicht werden können. Andere Darstellungen sprechen hier auch von *Nachfahren* und *Kindern*.

Es gibt zwei naheliegende Möglichkeiten, Graphen in einem Computerprogramm zu repräsentieren:

- Jeder einzelne Knoten ist ein Objekt, das in einem Array speichert, mit welchen anderen Knoten es verbunden ist. Ein solches Array wird *Nachbarschaftsliste* genannt.
- Alternativ lassen sich die Verbindungen zwischen sämtlichen Knoten eines Graphen auch in einer *Nachbarschaftsmatrix* speichern. Jeder Knoten wird durch eine Spalte und eine Zeile repräsentiert. Bei ungewichteten Graphen ist die Matrix zunächst mit Nullen gefüllt. Anschließend wird für jede Verbindung eine Eins eingetragen. Bei gewichteten Graphen stehen statt Einsen die Gewichte der jeweiligen Verbindungen in den Zellen. Die Abbildung zeigt das anhand des Beispiels mit den Rezeptzutaten aus Abschnitt 4.3, »Ein Beispiel von Hand gerechnet«:

Persönliches Exemplar für Pit Noack



Welche dieser beiden Repräsentationen besser geeignet ist, hängt von der Anwendung ab. Wenn wir beispielsweise nicht wissen, wie viele Knoten es überhaupt gibt, funktioniert nur die Nachbarschaftsliste. In Kapitel 2, »Texte bauen mit Markow«, haben wir den Übergangsgraphen des Markow-Prozesses in dieser Form programmiert. Im Fall der Assoziationsanalyse in Kapitel 4, »Wörter gruppieren«, hat sich die Nachbarschaftsmatrix als zweckmäßiger erwiesen.

In diesem Kapitel nutzen wir eine weitere Variante, wie in Abbildung 5.6 gesehen: Ein Algorithmus durchläuft den Suchbaum schrittweise per *Warteschlangenverfahren*, also einen Spielzustand nach dem anderen. Einmal besuchte Spielzustände (Knoten) werden nicht mehr benötigt und müssen daher nicht gespeichert werden. *Der Suchbaum ist daher niemals als Ganzes im Arbeitsspeicher vorhanden.*

5.3 Eine vielseitig einsetzbare Spiel-KI

Das Spiel »Fruchtkräsch« erfüllt drei Bedingungen, die Voraussetzung für unser Lösungsverfahren sind:

- Es ist ein vollständig *determiniertes* Spiel – das heißt, dass Zufallsentscheidungen wie Würfel- oder Münzwürfe keine Rolle spielen. Die Folgen eines Zugs hängen ausschließlich vom Zustand des Spielfelds und dem jeweiligen Zug ab.
- Es bietet *perfekte Information* – das bedeutet vereinfacht gesagt, dass der aktuelle Zustand des Spiels vollständig bekannt ist. Es gibt keine verdeckten Spielkarten wie beim Poker und keine verborgenen Zustände wie bei »Schiffe versenken«.
- Es ist *endlich*, das bedeutet, dass jede Partie nach einer endlichen Anzahl von Zügen vorbei ist.

Die hier vorgestellte KI kann im Prinzip jedes endliche zugbasierte Ein-Personen-Spiel mit perfekter Information und ohne Zufallselemente spielen. Damit die KI derart viel-

© Rheinwerk Verlag, Bonn 2023

seitig einsetzbar ist, müssen Spiellogik und KI weitgehend voneinander *entkoppelt* sein: Die KI weiß nichts über Spielfeld und Spielregeln. Sie kennt von einem Spielzustand lediglich *Bewertung* und *Zugmöglichkeiten*. Sie darf Züge ausprobieren und damit neue Spielzustände erzeugen und wiederum deren Bewertungen auslesen. Das reicht der KI, um einen Suchbaum wie den in Abbildung 5.2 zu durchlaufen und vielversprechende Züge zu finden.

Das, was die KI vom Spielzustand wissen muss, nennen wir die *Schnittstelle* zwischen Spiellogik und KI. Jedes Spiel, das die entsprechende Schnittstelle bietet, kann im Prinzip von der KI gespielt werden, ohne dass auch nur eine einzige Zeile Code in der KI geändert werden muss.

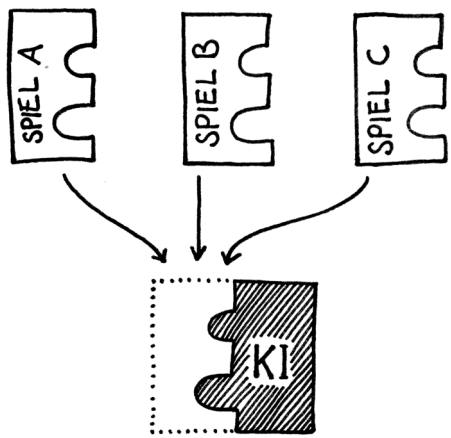


Abbildung 5.3 Unsere KI kann jedes Spiel spielen, das die entsprechende Schnittstelle bietet.

Im Folgenden konzentrieren wir uns auf jene Eigenschaften und Funktionen von Fruchtkräsch, die diese Schnittstelle bilden. Wie das Spiel gleichfarbige Gruppen findet, gültige Züge erkennt oder das Aufrücken der Spielsteine nach links und nach unten bewerkstelligt, ist nicht Thema dieses Kapitels. Falls Sie darüber mehr erfahren wollen, hilft Ihnen ein Blick in den ausführlich kommentierten Quellcode.

5.4 Die Klasse Spielzustand

Die komplette Fruchtkräsch-Spiellogik und die Schnittstelle für die KI stecken in der Klasse `Spielzustand`, zu finden in der Datei *code/fruchtkraesch.js*. Objekte dieser Klasse repräsentieren jeweils einen bestimmten Zustand des Spiels – das ist im Wesentlichen das Spielfeld mit den darin positionierten Spielsteinen.

Persönliches Exemplar für Pit Noack

5.4.1 Züge liefern modifizierte Spielzustände

Die Funktion `zug()` erwartet als Argument die Koordinaten eines Spielsteins in der Form `[zeile, spalte]`. Sofern der Zug gültig ist, *liefert die Funktion ein neues, modifiziertes* `Spielzustand`*-Objekt*; ist ein Zug ungültig, gibt die Funktion den ursprünglichen, unveränderten Spielzustand zurück:

```
spielzustand = spielzustand.zug([2, 4]);
```

Es gibt also nicht das *eine* Spiel, sondern nur einzelne Spielzustände – und Züge, die darauf angewendet werden und neue, modifizierte Spielzustände zurückliefern.

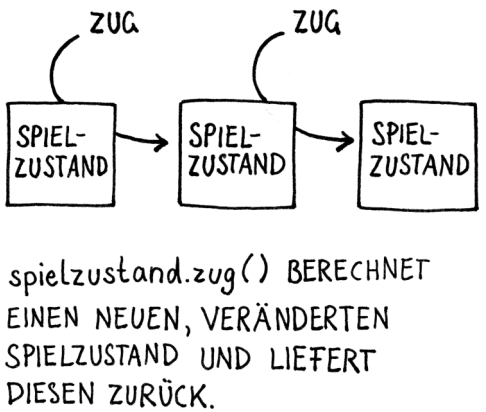


Abbildung 5.4 Jeder Spielverlauf ist eine Folge von Spielzuständen.

5.4.2 Die möglichen Züge

Jeder Spielzustand weiß, welche Züge auf ihm erlaubt sind. Wann immer ein neuer Spielzustand entsteht, werden die möglichen Züge berechnet und in einem Array mit dem Namen `moeglicheZuege` gespeichert.

5.4.3 Die Bewertung eines Spielzustands

Die Funktion `bewertung()` liefert eine ganze Zahl. Je höher die Zahl, umso besser ist der Spielzustand. Die KI benötigt die Bewertungen, um unterschiedliche Spielzustände miteinander zu vergleichen.

Wie in Abschnitt 5.2, »Wie findet die KI den besten Zug?«, beschrieben, können wir uns bei der Bewertung nicht allein auf den Punktestand verlassen, da wir auch Spielzustände beurteilen müssen, die noch Zugmöglichkeiten bieten. In unserer Lösung setzt sich die Bewertung daher aus dem Punktestand und dem *Potential* zusammen. So nennen wir

© Rheinwerk Verlag, Bonn 2023

die Anzahl aller Steine, die sich wegklicken lassen, sprich: zu irgendeiner gleichfarbigen Gruppe gehören (siehe Abbildung 5.5). Die Berechnung des Potentials nimmt vereinfachend an, dass sich durch einen Zug die Zugmöglichkeiten nicht ändern würden. Das trifft zwar nicht zu, hat sich in der Praxis aber als gute Näherungslösung bewährt.

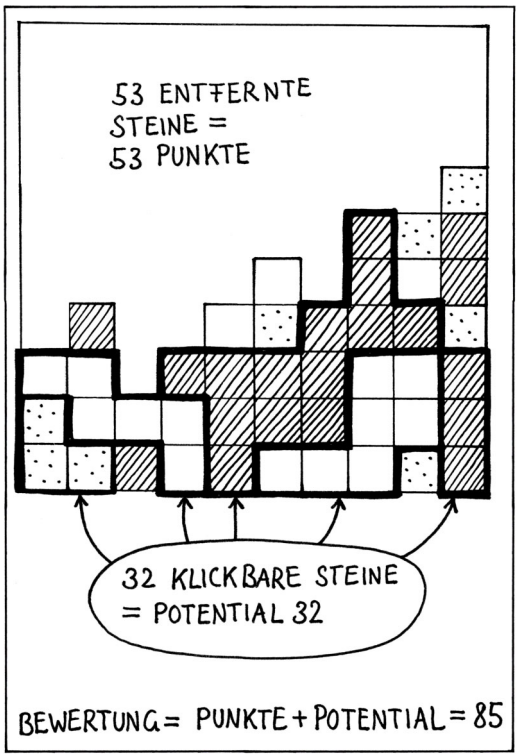


Abbildung 5.5 Die Bewertung eines Spielzustands ergibt sich aus dem Punktestand und dem Potential.

5.4.4 Ein Gedächtnis für Züge

Damit die KI rückverfolgen kann, welcher Zug zu dem am besten bewerteten Spielzustand geführt hat, muss es ein Gedächtnis für die vorherigen Züge geben. Dieses Gedächtnis heißt bei uns `zugArchiv`.

5.4.5 Die Schnittstelle im Überblick

Damit ist die Schnittstelle, welche die KI benötigt, um Fruchtkräsch zu spielen, vollständig beschrieben. Tabelle 5.1 fasst sie noch einmal zusammen.

Persönliches Exemplar für Pit Noack

Name	Aufgabe
`zug(zugKoord)`	Führt den gegebenen Zug durch und liefert einen neuen, veränderten Spielzustand zurück.
`moeglicheZuege`	Ein Array mit allen für diesen Spielzustand gültigen Zügen.
`zugIstMoeglich()`	Gibt `true` zurück, wenn mindestens ein Zug möglich, sonst `false`; Abkürzung für `moeglicheZuege.length > 0`.
`bewertung()`	Bewertet den Spielzustand und gibt dessen Wert zurück. Je höher der Wert, umso besser der Spielzustand.
`zugArchiv`	Ein Array, das vergangene Züge speichert.

Tabelle 5.1 Eigenschaften und Funktionen des Spielzustand-Objekts, die für die KI von Interesse sind

5.5 Die Klasse KI

Von Objekten der Klasse `KI` wollen wir nur eine einzige Sache wissen: *Was ist für einen gegebenen Spielzustand der vielversprechendste Zug?* Diese Frage beantworten Objekte der Klasse `KI` mit der Funktion `besterZug()`. Diese erwartet zwei Argumente: `spielzustand` und `suchtiefe`.

Alle weiteren Funktionen von `KI`-Objekten dienen lediglich dem internen Gebrauch und werden von `besterZug()` aufgerufen. Diese schauen wir uns der Reihe nach an.

5.5.1 Alle Folgezustände eines Spielzustands berechnen

Damit die KI einen Suchbaum konstruieren kann, muss sie viele Spielzustände untersuchen und alle jeweils möglichen Züge ausprobieren. Das erledigt die Funktion `berechneFolgezustaende()`. Sie erwartet als Argument einen Spielzustand und gibt ein Array zurück, das für jeden erlaubten Zug das Resultat – wiederum einen Spielzustand – speichert.

```
berechneFolgezustaende(spielzustand) {
  const moeglicheZuege = spielzustand.moeglicheZuege;
  return moeglicheZuege.map(einZug => spielzustand.zug(einZug));
}
```

Listing 5.1 Diese Funktion führt alle gültigen Züge aus und liefert ein Array mit den Ergebnissen.

© Rheinwerk Verlag, Bonn 2023

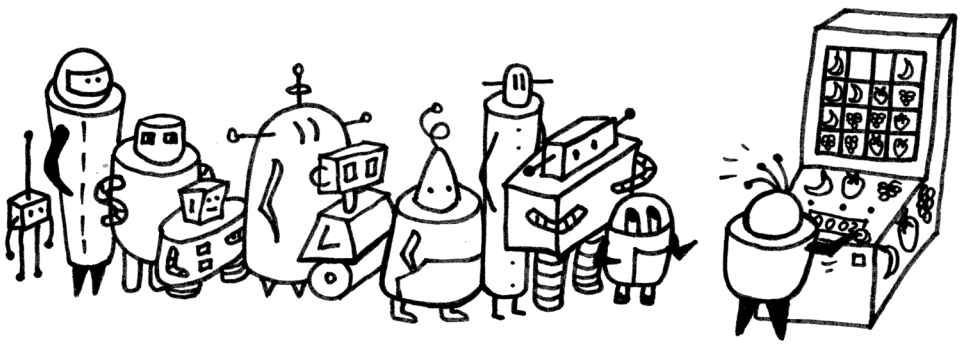
Falls Ihnen nicht klar ist, was in der letzten Zeile des Funktionskörpers passiert, schauen Sie sich bitte den Abschnitt A.13, »Funktionen höherer Ordnung« in Anhang A an.

Für die KI macht es übrigens keinen Unterschied, ob die Züge Koordinaten, Fließkommazahlen, Strings oder gar Objekte einer selbst programmierten Klasse sind. *Was ein gültiger Zug ist und was nicht, liegt allein in der Verantwortung des Spielzustands!* Die KI fragt lediglich die gültigen Züge ab, wendet sie systematisch an und sammelt die Ergebnisse. Diese getrennten Verantwortlichkeiten sind Voraussetzung für die in Abschnitt 5.3 beschriebene vielseitige Anwendbarkeit der Spiel-KI.

5.5.2 Alle Spielverläufe per Warteschlangenverfahren berechnen

Die eben beschriebene Funktion ist ein wichtiger Baustein für die Berechnung des Suchbaums. Diese erledigt die Funktion `berechneSpielverlaeufe()`. Genau wie `besterZug()` erwartet sie zwei Argumente – einen Spielzustand und eine Suchtiefe. Rückgabewert ist ein Array mit Spielzuständen, bei denen die Suchtiefe erreicht wurde *oder* kein Zug mehr möglich ist.

Die Funktion nutzt eine *Warteschlange*, um den Baum zu durchsuchen. Eine solche Schlange funktioniert genau so, wie wir es aus dem Alltag kennen. Wer sich zuerst anstellt, kommt auch als Erstes dran. Dies wird als *First In First Out*-Prinzip (kurz *FIFO*) bezeichnet. Die Warteschlange heißt auch im Deutschen meist *Queue*.



```
berechneSpielverlaeufe(spielzustand, suchtiefe) {
  let warteschlange = [spielzustand];
  let resultat = [];

  while (warteschlange.length != 0) {
    const spielzustand = warteschlange.shift();

    // keine Züge mehr möglich
    if (!spielzustand.zugIstMoeglich()) {
      resultat.push(spielzustand);
      continue;
    }

    // Suchtiefe erreicht
    if (spielzustand.zugArchiv.length == suchtiefe) {
      resultat.push(spielzustand);
      continue;
    }

    // Züge möglich und Suchtiefe nicht erreicht
    let folgezustaende = this.berechneFolgezustaende(spielzustand);
    this.sortiereNachBewertung(folgezustaende);
    // Beschneidung des Suchbaums
    folgezustaende = folgezustaende.slice(0, 3);
    warteschlange = warteschlange.concat(folgezustaende);
  }

  return resultat;
}
```

Listing 5.2 Die Funktion ki.berechneSpielverlaeufe() durchkämmt den Suchbaum mittels einer Warteschlange.

Das Listing 5.2 zeigt den kompletten Code. Die erste Zeile des Funktionskörpers initialisiert die Warteschlange. Diese enthält zunächst lediglich den per Argument übergebenen Spielzustand, also den Wurzelknoten des Baums, den die Funktion durchsuchen soll: `[spielzustand]`. Das Array `resultat` ist der Behälter für alle »fertigen« Spielzustände – also jene, die die Suchtiefe erreicht haben oder keine Zugmöglichkeiten mehr bieten.

Die `while`-Schleife läuft so lange, bis die Warteschlange keinen Spielzustand mehr enthält. Innerhalb der Schleife passiert Folgendes:

© Rheinwerk Verlag, Bonn 2023

- `shift()` entnimmt der Warteschlange den vorne stehenden `spielzustand` zur weiteren Betrachtung.
- Wenn `spielzustand` keine Zugmöglichkeiten bietet, wird dieser dem Array `resultat` hinzugefügt. Der aktuelle Durchgang der `while`-Schleife wird dann per `continue` abgebrochen.
- Das Gleiche passiert, wenn `spielzustand` die vorgegebene `suchtiefe` erreicht hat.
- Andernfalls werden mittels `berechneFolgezustaende()` alle auf dem `spielzustand` möglichen Züge durchgeführt.
- Die Funktion `sortiereNachBewertung()` sortiert die `folgezustaende`, sodass die am besten bewerteten vorne stehen.
- `slice()` wählt die ersten drei Zustände aus, und `concat()` hängt diese an die Warteschlange an. Hier werden also nicht alle Spielzustände weiterverfolgt, sondern nur drei besonders vielversprechende. Das ist das in Abschnitt 5.2, »Wie findet die KI den besten Zug?«, angesprochene Pruning, also die Beschneidung des Suchbaums.
- Sofern `warteschlange` noch Spielzustände enthält, wird der Körper der `while`-Schleife erneut durchlaufen.
- Andernfalls liefert die Funktion das Array mit den Ergebnissen zurück.

Abbildung 5.6 stellt den Weg durch den Suchbaum per Warteschlange dar.

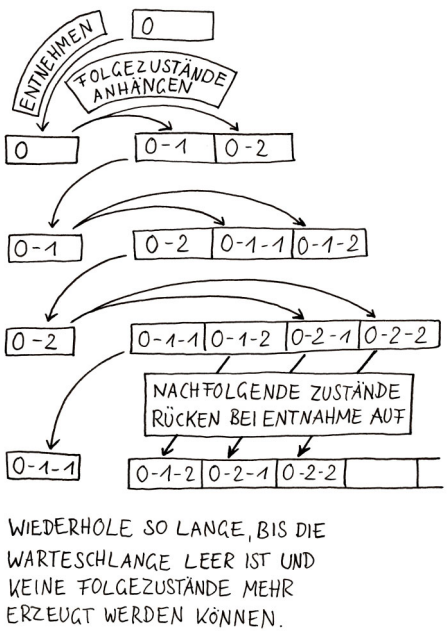


Abbildung 5.6 Berechnung der Spielverläufe mittels Warteschlangenverfahren

5.5.3 Die Spielzustände nach Bewertung sortieren

Mit den jeweils besten Spielzuständen soll weitergearbeitet werden. Dafür müssen diese nach der Bewertung sortiert werden. Das erledigt die Funktion `sortiereNachBewertung()`. Sie erwartet als Argument ein Array mit Spielzuständen und sortiert diese mittels `sort()` abhängig vom Rückgabewert der Funktion `bewertung()`. Höher bewertete Spielzustände stehen weiter vorn. Wohlgemerkt, es ist nicht die Aufgabe der KI, diese Bewertung durchzuführen, der Spielzustand selbst ist dafür zuständig.

```
sortiereNachBewertung(spielzustaende) {
  spielzustaende.sort((spielzustandL, spielzustandR) =>
    spielzustandR.bewertung() - spielzustandL.bewertung());
}
```

Listing 5.3 Die Sortierung von Spielzuständen nach deren Bewertung

5.5.4 Die Funktion besterZug()

Zu guter Letzt schauen wir uns den Code der Funktion `besterZug()` an. Die Funktion vollzieht das, was Abbildung 5.2 darstellt: Wir suchen den Spielzustand mit der höchsten Bewertung. Der erste Zug, der in Richtung dieses Endzustands führt, ist der beste Zug.

```
besterZug(spielzustand, suchtiefe) {
  spielzustand.zugArchiv = [];

  const zustaende = this.berechneSpielverlaeufe(spielzustand, suchtiefe);
  this.sortiereNachBewertung(zustaende);
  const besterZustand = zustaende[0];
  const besterZug = besterZustand.zugArchiv[0];
  return besterZug;
}
```

Listing 5.4 Die Funktion ki.besterZug()

Zuerst muss das Zuggedächtnis des untersuchten Spielzustands gelöscht werden. Denn die KI interessiert sich nicht für vergangene, bereits ausgeführte Züge, sondern allein für die vom aktuellen Spielzustand aus möglichen künftigen Spielverläufe. Die Anzahl der Einträge im Zugarchiv wird auch zur Kontrolle der Suchtiefe genutzt.

Ein Aufruf der Funktion `berechneSpielverlaeufe()` liefert das Array `zustaende` mit den Spielzuständen, die keine Zugmöglichkeiten mehr bieten oder die Suchtiefe erreicht haben. Anschließend wird `zustaende` absteigend nach Bewertung sortiert. Der beste Spiel-

© Rheinwerk Verlag, Bonn 2023

zustand steht ganz vorne im Array: `besterZustand = zustaende[0]`. Der beste Zug ist dann der erste im Zugarchiv des ausgewählten Spielzustandes: **`besterZug = besterZustand.zugArchiv[0]`**.

5.5.5 Die Funktionen des KI-Objekts im Überblick

Tabelle 5.2 fasst noch einmal alle Funktionen des KI-Objekts zusammen.

Name	Aufgabe
`besterZug(spielzustand, suchtiefe)`	Berechnet den bestmöglichen Zug ausgehend von `spielzustand` bei entsprechender `suchtiefe`.
`berechneFolgezustaende(spielzustand)`	Führt alle für `spielzustand` gültigen Züge aus und liefert ein Array mit den resultierenden Spielzuständen.
`berechneSpielverlaeufe(spielzustand, suchtiefe)`	Berechnet alle Spielverläufe ausgehend vom `spielzustand`. Der Rückgabewert ist ein Array mit allen Spielzuständen, bei denen die `suchtiefe` erreicht wurde oder keine Züge mehr möglich sind.
`sortiereNachBewertung(spielzustaende)`	Sortiert ein Array mit Spielzuständen absteigend nach Bewertung.

Tabelle 5.2 Funktionen des KI-Objekts

5.6 Ideen zum Weitermachen

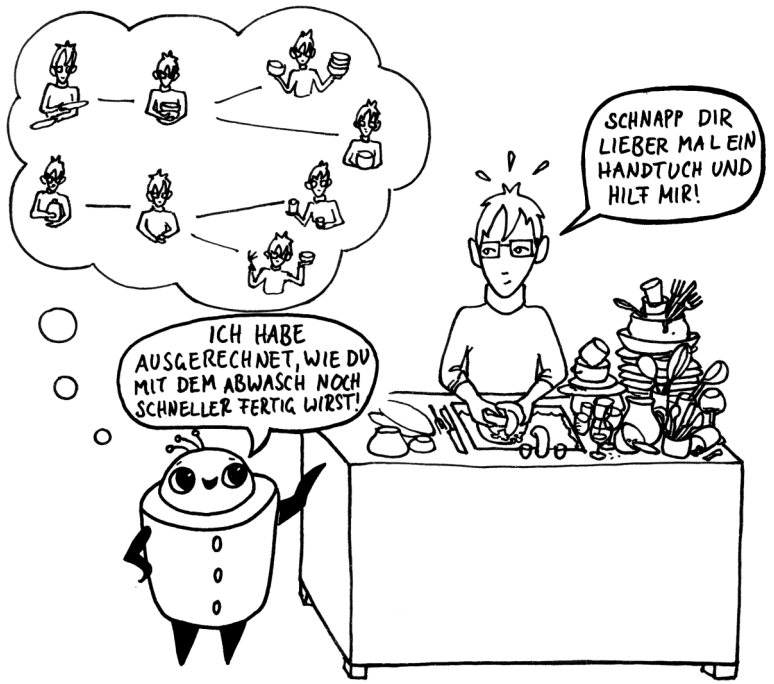
Ein eigenes Spiel zu programmieren und dieses mittels der beschriebenen Schnittstelle an die KI anzuschließen – das ist eine reizvolle, aber anspruchsvolle Aufgabe. Dennoch können Sie hier eine Menge lernen, selbst wenn Sie ein solches Projekt nur mit Papier und Bleistift skizzieren.

Überlegen Sie, welche Spiele außer Fruchtkräsch geeignet sein könnten, um sie mit der hier vorgestellten KI zu spielen. Wenn Sie ein geeignetes Spiel gefunden haben, können Sie sich über die entscheidende Frage nachdenken, wie die Bewertungsfunktion beschaffen sein müsste.

Vielleicht schauen Sie sich bei der Gelegenheit das Spiel »Rush Hour« an. Das ist ein Ein-Personen-Spiel mit perfekter Information und ohne Zufallselemente. Können Sie sich

Persönliches Exemplar für Pit Noack

vorstellen, dafür eine Bewertungsfunktion zu schreiben, die unfertige Spielzustände sinnvoll beurteilen kann?



5.7 Zusammenfassung und Ausblick

Die in diesem Kapitel vorgestellte KI leistet innerhalb der beschriebenen Grenzen Beachtliches. Im Hinblick auf das Spiel »Fruchtkräsch« haben wir den Eindruck gewonnen: Wenn es eine perfekte Lösung ohne übrig gebliebene Steine gibt, wird die KI diese Lösung bei hinreichender Suchtiefe auch finden. Sollte eine Leserin oder ein Leser ein Gegenbeispiel finden, freuen wir uns über Zuschriften.

Sobald Ungewissheit in Form von Zufallsereignissen oder verdeckten Zuständen ins Spiel kommt, versagt unsere KI kläglich. Bei ähnlich gestrickten Spielen wie »Candy Crush« oder »Collapse« rücken bei der Entfernung von Steinen aus dem Bereich jenseits des oberen Spielfeldrands neue Steine mit zufälligen Farben nach. Ein solches Spiel kann unsere KI nicht lösen. Auch hier laden wir Sie dazu ein, darüber nachzudenken, wie eine Lösung aussehen könnte und mit welchen Problemen dabei zu rechnen ist. Wir werden diese spannende Frage in diesem Buch nicht behandeln, sondern wenden uns im nächsten Spiel einer KI zu, die Zwei-Personen-Spiele mit perfekter Information meistert.

© Rheinwerk Verlag, Bonn 2023