Bäume und Backtracking

Leitprogramm
Harald Pierhöfer

Inhalt:
Dieses Leitprogramm führt in das Thema von Bäumen und Backtracking ein und richtet sich an Gymnasialschülerinnen und -schüler im letzten Schuljahr vor der Matur.

Unterrichtsmethode: Leitprogramm

Fachliches Review:
Juraj Hromkovic, Informationstechnologie und Ausbildung, ETH Zürich

Fachdidaktisches Review:
Juraj Hromkovic, Informationstechnologie und Ausbildung, ETH Zürich
Giovanni Serafini, Informationstechnologie und Ausbildung, ETH Zürich

Publiziert auf EducETH:
20. Juni 2011

Leitprogramm zu

Bäume und Backtracking

Mentorierte Arbeit
Harald Pierhöfer
Winter 09

Vorwort

Dieses Leitprogramm führt in das Thema von Bäumen und Backtracking ein und richtet sich an Gymnasialschülerinnen und -schüler im letzten Schuljahr vor der Matur. Die folgenden Voraussetzungen müssen gegeben sein:

- Es wird auf der Baumsichtweise aufgebaut, wie sie in der Wahrscheinlichkeitsrechnung gebraucht wird. In das Thema der Wahrscheinlichkeitsrechnung sollte im Mathematik-Unterricht deshalb schon eingeführt worden sein.
- Der Begriff des ungerichteten Graphen sollte bekannt sein. Eine Zusammenfassung dieses Themas ist im Anhang gegeben.
- Die Lernenden benötigen Kenntnisse in einer Programmiersprache. (Der Code in diesem Leitprogramm ist in Processing angegeben, nähere Informationen dazu unter `www.processing.org`). Neben den Grundlagen muss auch die rekursive Programmierung verstanden worden sein.

Zur Bearbeitung sind mindestens 15 Lektionen (plus entsprechende Zeit für die Arbeit zu Hause als Hausaufgaben) vorgesehen. In dieser Zeit wird es allerdings nicht allen Schülerinnen und Schüler möglich sein, sämtliche Aufgaben aus dem letzten Kapitel zu bearbeiten, da die einzelnen Aufgaben zeitlich aufwändig sind.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Wie kann hier gearbeitet werden?

Dieser Text wird Sie mit einer vielseitig einsetzbaren Problemlösungsstrategie bekannt machen: dem Backtracking. Schon im nächsten Unterkapitel finden Sie eine Einführung in das Thema, doch bevor Sie loslegen, soll Ihnen hier noch kurz der Aufbau des Textes und die Arbeitsweise damit erläutert werden.

Die folgenden Kapitel sind zur selbstständigen Bearbeitung gedacht. Sie werden in die Theorie von gerichteten Graphen und Bäumen eingeführt, lernen, wie Sie Suchbäume programmieren können, und werden dann das Backtracking-Verfahren kennen lernen. Wenn Sie von diesen Begriffen jetzt noch keine Ahnung haben, macht das nichts. Mitbringen sollten Sie aber die folgenden Kenntnisse:

Voraussetzungen für die Bearbeitung dieses Texts

- Sie wissen, was ein Algorithmus ist.
- Sie haben Kenntnisse einer Programmiersprache. (Die hier vorgestellten Codes sind in Processing geschrieben; einer Sprache, die sich stark an Java anlehnt)
- Sie kennen die rekursive Programmierung, in welcher sich Funktionen selbst wieder aufrufen.
- Sie kennen die Grundbegriffe ungerichteter Graphen. Sollte dies nicht der Fall sein, können Sie die Zusammenfassung im Anhang A.2 durchlesen, in welcher das Wichtigste kurz erklärt wird.

Zur behandelten Theorie finden Sie immer auch Aufgaben, anhand derer Sie das Gelernte prüfen können. Die Lösungen dieser Aufgaben stehen jeweils im zweitletzten Unterkapitel eines jeden Themas. Das letzte Unterkapitel ist dann der Kapiteltest, den Sie bearbeiten und mit Ihrer Lehrperson besprechen sollten.

1.2 Worum geht es hier?

Wahrscheinlich kennen Sie die Rätselart der Sudoku, die in vielen Tageszeitungen und Rätselheften zu finden ist, und vielleicht haben Sie ja auch schon eines gelöst. Die Ausgangslage besteht aus einigen Zahlen von 1 bis 9, die in einem quadratischen Gitter von 9 auf 9 Feldern angeordnet sind. Die Aufgabe ist es, weitere solche Zahlen in die noch leeren Felder zu füllen, so dass in jeder Zeile, in jeder Spalte und in jedem der hervorgehobenen 3×3 Untertableaux jede Zahl genau einmal vorkommt.

			1	7			2	
2					9	7		
	7		2	8	3	4		
	5	7						
3		9				8		7
						5	3	
		8	3	1	6		4	
		5	7					8
	4			5	8			

Diese Aufgabenstellung ist ein typischer Vertreter von Problemen, die sich mit der Methode des Backtrackings lösen lassen. Die folgenden Seiten werden Ihnen zeigen, wie dies geht. Sehen wir uns zuvor jedoch noch die Struktur des Sudoku-Problems an: Die Aufgabenstellung heisst:

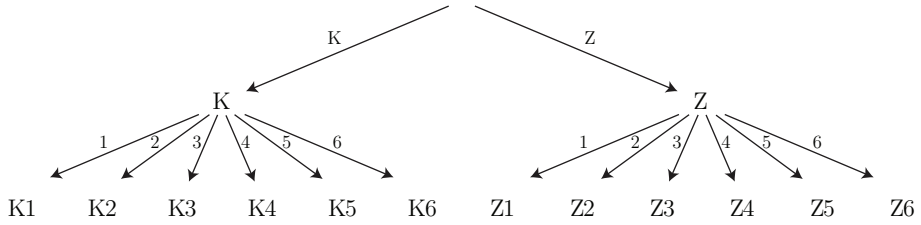
1. Füllen Sie in jedes leere Kästchen genau eine Zahlen von 1 bis 9 ein.
2. Tun Sie dies unter der Bedingung, dass in jeder Zeile, jeder Spalte und jedem 3×3-Untertableaux jede Zahl von 1 bis 9 genau einmal vorkommt.

Der erste Punkt gibt eine Menge von Möglichkeiten vor. Jedes Element dieser Menge entspricht einer Belegung aller Kästchen mit bestimmten Zahlen von 1 bis 9. Im obigen Beispiel gibt es 51 leere Kästchen und daher $9^{51} = 5 \cdot 10^{48}$ Möglichkeiten, diese zu belegen. Der zweite Punkt ist meist in nur einer einzigen dieser Möglichkeiten erfüllt, der Lösung. Es wäre jedoch auch vorstellbar, dass es mehrere Lösungen oder gar keine Lösung gibt. Es stellt sich die Frage, wie aus der (unter Umständen sehr grossen) Menge der Möglichkeiten die Lösung(en) gefunden werden können. Dabei können primär zwei Wege unterschieden werden: man könnte versuchen, in einem Programm die Regeln zu implementieren, wie deduktiv auf die Belegung eines Feldes geschlossen werden kann, und so schrittweise das ganze Sukoku zu lösen versuchen. Dies ergäbe einen wahrscheinlich ziemlich schnellen und natürlich auch sehr spezifischen Algorithmus, der das Vorgehen eines guten Sudoku-Lösers imitieren würde. Die Nachteile dieses Ansatzes sind, dass die Programmierung aufwändig ist und auch nicht sichergestellt ist, dass das Programm die Lösung in jedem Fall findet. In ähnlichen

Problemstellungen könnte es auch sein, dass es überhaupt kein Vorgehen gibt, das deduktiv auf Lösungen schliesst.

Wir verfolgen hier daher einen anderen, ziemlich allgemeinen Ansatz. Nacheinander generieren wir alle Möglichkeiten und überprüfen jeweils, ob eine auch die zusätzliche Bedingung (Punkt 2 oben) erfüllt. Ist dies der Fall, können wir aufhören (ausser wir wollen überprüfen, ob es weitere Lösungen gibt), andernfalls nehmen wir die nächste Möglichkeit. Natürlich müssen wir uns Gedanken darüber machen, wie wir dieses unter Umständen sehr zeitintensive Vorgehen abkürzen können, und das werden wir im Kapitel 4 auch tun. Zuvor aber setzen wir uns mit Bäumen auseinander, denn sie ermöglichen es uns, die Menge der Möglichkeiten zu generieren und darin eine Reihenfolge zu finden, in welcher sie abgearbeitet werden sollen.

Sie kennen Bäume bereits aus der Wahrscheinlichkeitsrechnung. Wir werden später wieder auf das Sukoku zu sprechen kommen, da die Mächtigkeit seiner Möglichkeitenmenge zu gross ist, sehen wir uns das Beispiel eines Münzwurfs an, dem ein Wurf eines Würfels folgt. Die Ergebnismenge dieses Zufallsexperiments entspricht der Menge aller möglichen Ausgänge. Wir können sie mit Hilfe eines Baums veranschaulichen. Solche Bäume haben Sie eventuell schon in der Wahrscheinlichkeitsrechnung kennen gelernt. Sie werden dort eingesetzt, um die Ergebnismenge mehrstufiger Zufallsexperimente zu veranschaulichen.



Beschäftigen wir uns also vorerst mit der Frage, was ein Baum genau ist, und wie er beschrieben werden kann.

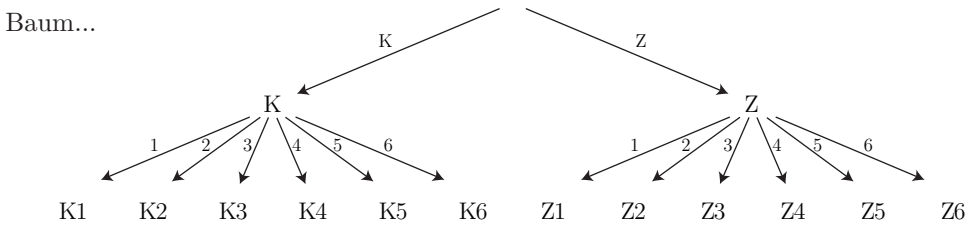
Kapitel 2

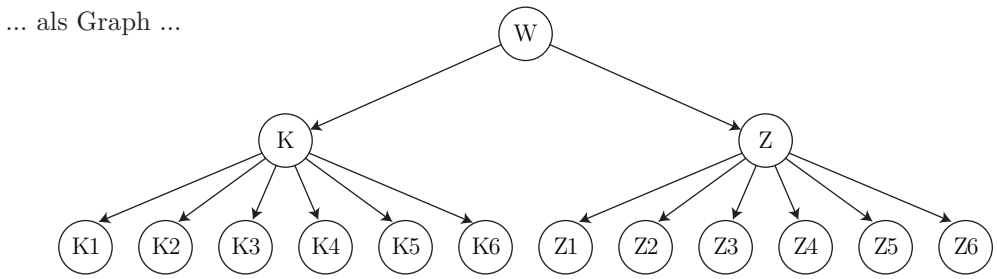
Bäume

Wir werden Bäume brauchen, um den ganzen Möglichkeitenraum eines Problems aufzuspannen. Da Bäume spezielle Graphen sind, können Sie sich in diesem Kapitel das nötige Wissen aneignen, um Bäume adäquat zu beschreiben.

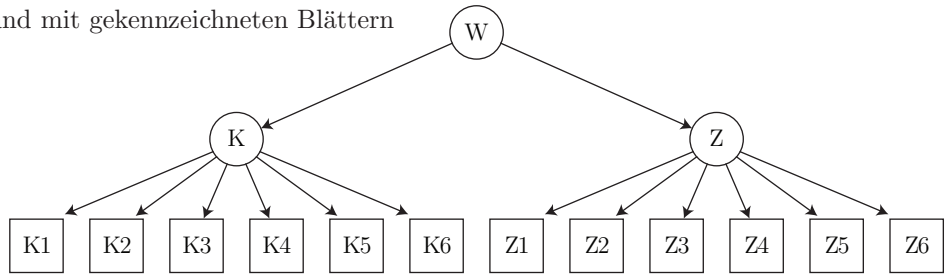
2.1 Was hat ein Baum mit einem Graphen zu tun?

Bäume gibt es in der Informatik in vielen Bereichen, als grafische Darstellung oder auch als Datenstruktur. Wir betrachten hier die grafische Darstellung, wenn ausgehend von einem Anfangspunkt sich der Baum (eventuell wiederholt) verzweigt, bis alle Äste vollständig vorhanden sind. Den Anfangspunkt nennen wir die **Wurzel**, an den Enden der Äste zeichnen wir je ein **Blatt**. Die Wurzel, die Verzeigungspunkte und die Blätter können wir als Knoten sehen, die vorhandenen Verbindungslinien als Kanten, womit wir sehen, dass ein Baum auch ein Graph ist. Alle Knoten, die weder Wurzel noch Blatt sind, nennen wir **innere Knoten**.



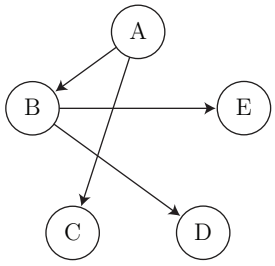


... und mit gekennzeichneten Blättern



Die Kanten dieses Graphen haben eine Richtung, die von der Wurzel weg hin zu den Blättern zeigt. Im Graphen können wir dies sichtbar machen, indem wir statt Verbindungslinien nun Pfeile zeichnen. Es liegt dann ein **gerichteter Graph** vor. Weiter gibt es in einem Baum keine **Mehrfachkanten**, also niemals zwei Kanten, die vom gleichen Ausgangsknoten zum selben Endknoten verlaufen. Einen solchen gerichteten Graphen ohne Mehrfachkante fassen wir als $G(V, E)$ auf, aufgebaut aus einer Menge V von Knoten ("vertices") und einer Menge E von Kanten ("edges"), wobei E eine Teilmenge aus dem kartesischen Produkt $V \times V$ ist.

Das nebenstehende Beispiel zeigt einen gerichteten Graphen $G(V, E)$ mit $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ und $E = \{(\mathrm{A}, \mathrm{B}), (\mathrm{A}, \mathrm{C}), (\mathrm{B}, \mathrm{D}), (\mathrm{B}, \mathrm{E})\}$.

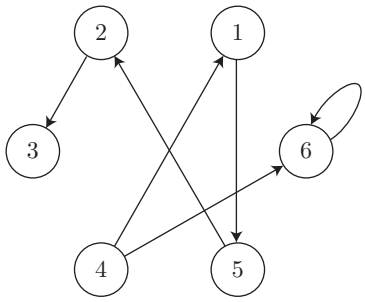


Aufgabe 2.1.1

Zeichnen Sie einen gerichteten Graphen $G(V, E)$ mit Knoten $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$ und Kanten $E = \{(\mathrm{A}, \mathrm{B}), (\mathrm{A}, \mathrm{D}), (\mathrm{C}, \mathrm{D}), (\mathrm{E}, \mathrm{A}), (\mathrm{F}, \mathrm{A})\}$.

Aufgabe 2.1.2

Geben Sie zu nebenstehendem Graphen $G = (V, E)$ die Menge V der Knoten und die Menge E der Kanten an.

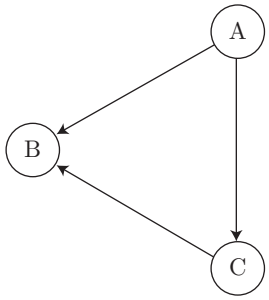


2.2 Eingangsgrad, Ausgangsgrad

In einem ungerichteten Graphen ist der **Grad** eines Knotens die Anzahl der Kanten, die an ihm ansetzen. In einem gerichteten Graphen nun wird unterschieden, ob eine Kante im Knoten eintrifft (dann erhöht sie den **Eingangsgrad** um 1) oder von ihm wegführt (wodurch der **Ausgangsgrad** um 1 erhöht wird.)

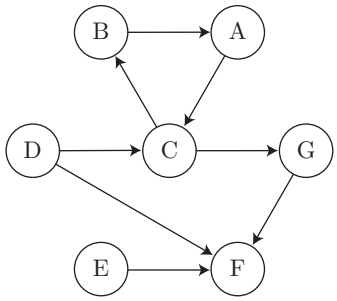
Beispiel

Im nebenstehenden Graphen hat der Knoten A den Eingangsgrad 0 und den Ausgangsgrad 2. Im Knoten C hingegen ist sowohl der Eingangs- als auch der Ausgangsgrad 1.



Aufgabe 2.2.1

Welcher Knoten hat den grössten Eingangsgrad, welcher den kleinsten Ausgangsgrad?



Aufgabe 2.2.2

In einem Graphen $G(V, E)$ ist die Menge der Knoten $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ und die Menge der Kanten ist $E = \{(\mathrm{C}, \mathrm{A}), (\mathrm{E}, \mathrm{B}), (\mathrm{D}, \mathrm{E}), (\mathrm{A}, \mathrm{D}), (\mathrm{B}, \mathrm{A})\}$.

Zeichnen Sie den Graphen und bestimmen Sie Ein- und Ausgangsgrad aller Knoten.

Aufgabe 2.2.3*

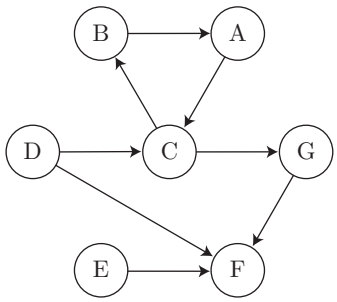
In einem Graphen $G(V, E)$ mit Ecken $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ sind alle Ein- und Ausgangsgrade gegeben:

Eingangsgrad(A) = 0, Ausgangsgrad(A) = 2;
Eingangsgrad(B) = 2, Ausgangsgrad(B) = 1;
Eingangsgrad(C) = 1, Ausgangsgrad(C) = 2;
Eingangsgrad(D) = 2, Ausgangsgrad(D) = 0;

Weiter ist bekannt, dass es keine Mehrfachkanten gibt (also keine Kante doppelt vorkommt) und zu keiner Kante (x, y) $(x, y \in V)$ die Gegenkante (y, x) existiert, insbesondere existiert auch keine Schlinge (x, x). Bestimmen Sie alle solche Graphen.

2.3 Weg, Pfad, Kreis und Zyklus

Stellen wir uns vor, dass wir auf einem Graphen wanderten. Dabei könnten wir natürlich nur auf den vorhandenen Kanten laufen; sind diese gerichtet, so handelt es sich um Einbahnstrassen. Im nebenstehenden Graphen könnten wir so von Knoten D zu C, von dort aus zu B, weiter zu A, wieder zu C und noch zu G gehen. Ein solcher Kantenzug wird **Weg** genannt. Einen Weg (auch Pfad genannt) kann man als Folge von Knoten darstellen, die miteinander über Kanten verbunden sind.



Etwas genauer:

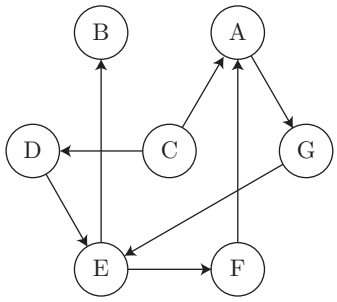
Ein Weg (Pfad) P in einem Graphen $G(V, E)$ ist eine Folge $(k_1, k_2, \ldots, k_n)$ mit $k_i \in V$ und $(k_i, k_{i+1}) \in E$, für alle $1 \leq i < n$.

Unserer oben beschriebener Weg ist also (D,C,B,A,C,G).

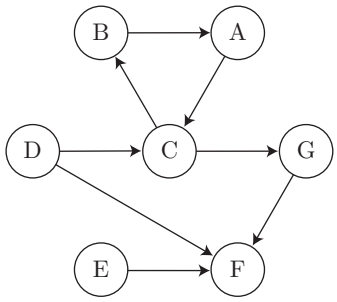
Die Länge des Weges ist die Anzahl der vorkommenden Kanten und damit die Anzahl der Knoten des Weges minus 1. Also hat der Weg (D,C,B,A,C,G) die Länge 5.

Aufgabe 2.3.1

Geben Sie den kürzesten Weg an von C nach B.



Häufig ist es nicht sinnvoll Wege zu gehen, in denen man im Kreis herumzuläuft. Im Weg (D,C,B,A,C,G) laufen wir von C über B und A wieder zu C. Das Teilstück (C,B,A,C) ist ein **Kreis**, denn Anfangs- und Endknoten sind gleich, die anderen Knoten aber alle verschieden. Lassen wir diesen Kreis weg, so erhalten wir mit (D,C,G) einen Weg, in welchem kein Knoten mehrfach vorkommt. Ein solcher Weg wird **einfacher Weg** genannt.

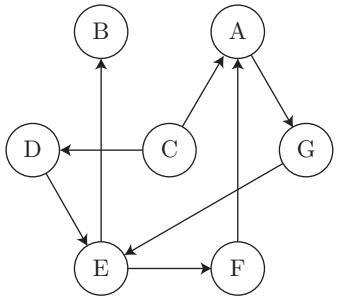


Ein Kreis ist ein Weg $(k_1, k_2, \ldots, k_n)$ mit $k_1 = k_n$ und $k_i \neq k_j,\ \forall i, j,\ 1 \leq i, j < n,\ i \neq j$

Ein einfacher Weg ist ein Weg $(k_1, k_2, \ldots, k_n)$ mit $k_i \neq k_j\ \forall i, j\ ,\ 1 \leq i, j \leq n,\ i \neq j$.

Aufgabe 2.3.2

Geben Sie alle einfachen Wege an von C nach A.

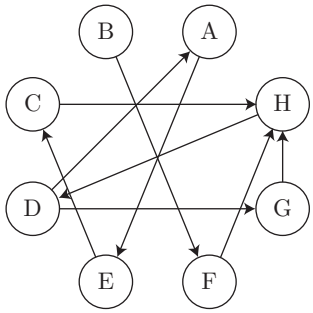


Ein **Zyklus** ist ein etwas verallgemeinerter Kreis. Wie in einem Kreis müssen in einem Zyklus Anfangs- und Endknoten identisch sein. Darüber hinaus ist es aber auch erlaubt, dass weitere Knoten mehrfach vorkommen.

Ein Zyklus ist ein Weg $(k_1, k_2, \ldots, k_n)$ mit $k_1 = k_n$.

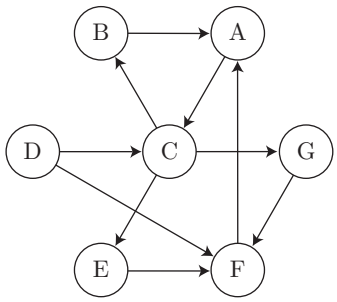
Aufgabe 2.3.3

a) Geben Sie alle von A ausgehenden Kreise an.
b) Durch welche Knoten geht der kürzeste Zyklus mit Startknoten A, der kein Kreis ist?



Aufgabe 2.3.4

a) Wie viele Wege und wie viele einfache Wege gibt es im folgenden gerichteten Graphen von A zu F?
b) Wie viele Kreise gibt es? (Wir betrachten es nicht als neuen Kreis, wenn die Knoten einfach zyklisch vertauscht werden. Ist beispielsweise (X,Y,Z,X) ein Kreis, so wird (Y,Z,X,Y) nicht als neuer, sondern als der selbe Kreis gezählt.)



Aufgabe 2.3.5*

Beweisen oder widerlegen Sie folgende Aussage: Gibt es in einem gerichteten Graphen einen Anfangs- und einen Endknoten, für die es mehr Wege als Pfade gibt, so gibt es unendlich viele Wege von diesem Anfangs- zum Endknoten.

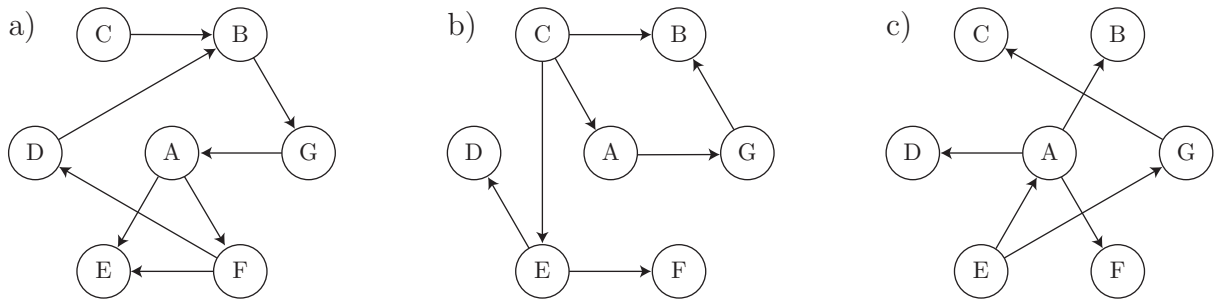
("Beweisen" bedeutet, die allgemeine Gültigkeit der Aussage in jedem Graphen zu begründen. "Widerlegen" heisst, einen Graphen zu finden, in dem die Aussage nicht stimmt.)

2.4 Alle Bäume sind Graphen - aber welche Graphen sind Bäume?

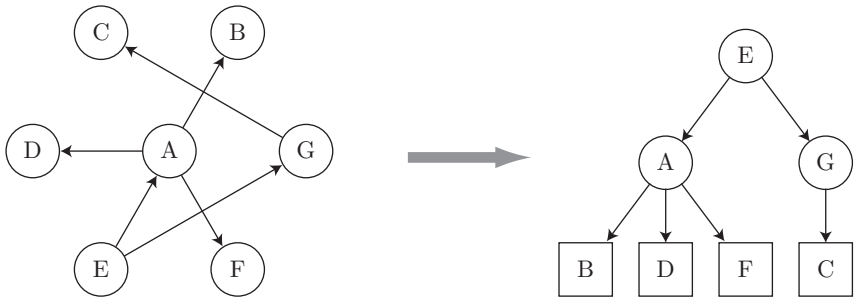
Ein Graph, welcher einen Zyklus enthält, wird **zyklisch** genannt. Enthält ein Graph keinen Zyklus, so ist er **azyklisch**.

Aufgabe 2.4.1

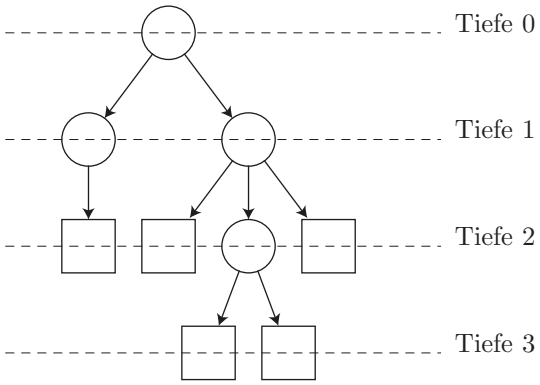
Welche der folgenden Graphen sind zyklisch, welche azyklisch? Gib für die zyklischen Graphen mindestens einen Zyklus an.



Wir haben Bäume schon als gerichtete Graphen ohne Mehrfachkanten identifiziert. Nicht jeder gerichtete Graph ohne Mehrfachkanten ist aber ein Baum, und auch wenn es einer ist, so ist dies in der graphischen Darstellung nicht unbedingt sofort erkennbar. Von den drei Graphen der letzten Aufgabe ist nur einer ein Baum. Der Graph c) kann auch wie folgt gezeichnet werden:



Sehen wir uns an, wie ein Baum gezeichnet werden kann. Dazu starten wir in einem Knoten mit Eingangsgrad 0 und nennen diesen die **Wurzel**. Von dort aus ziehen wir gerichtete Kanten zu weiteren Knoten, denen wir die Tiefe 1 zuordnen, da sie von der Wurzel aus mit einem Kantenzug der Länge 1 erreicht werden können. (In den Bäumen aus der Wahrscheinlichkeitsrechnung entspricht das den Ereignissen der ersten Stufe.) Von diesen Knoten aus können wir weitere Kanten zu Knoten der Tiefe 2 ziehen (Stufe 2), und so fort. Jeder Knoten mit Ausnahme der Wurzel wird von der Wurzel aus auf genau einem Weg erreicht, also ist der Eingangsgrad eines jeden Knotens mit Ausnahme der Wurzel gleich 1. Da alle gerichtete Kanten von Tiefe j zu Tiefe $j+1$ zeigen, kann es auch keine Zyklen geben.



Motiviert aus der obigen Beschreibung definieren wir einen Baum nun folgendermassen:

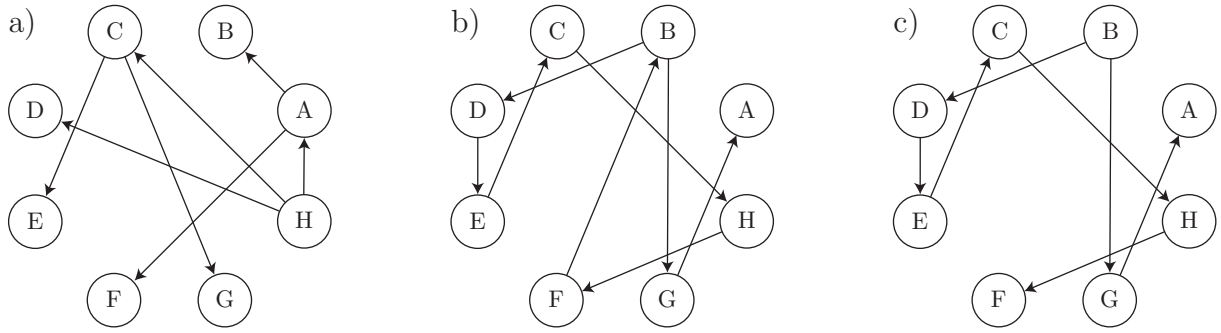
Ein Baum ist ein gerichteter, azyklischer Graph mit den folgenden Eigenschaften:

- Es gibt genau einen Knoten mit Eingangsgrad 0. Dieser wird Wurzel genannt.
- Alle anderen Knoten des Baums haben Eingangsgrad 1.

Diejenigen Knoten des Baums, die Ausgangsgrad 0 haben, bezeichnen wir als **Blätter** und zeichnen sie als Rechtecke. Alle Knoten, die weder Wurzel noch Blatt sind, nennen wir **innere Knoten**. Als **Höhe** eines Baums geben wir die maximale Tiefe seiner Knoten an. Es ist dies gleichzeitig die maximale Länge eines Pfades im Baum. Im oben gezeichneten Baum ist die maximale Tiefe gleich 3; die Höhe des Baums beträgt daher 3.

Aufgabe 2.4.2

Welche der folgenden Graphen sind Bäume? Zeichnen Sie die Bäume in der "Wurzel oben"-Darstellung und geben Sie die Höhe an.



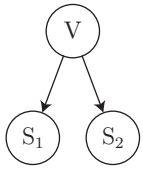
Aufgabe 2.4.3*

Geben Sie einen Algorithmus an, der überprüft, ob es sich bei einem gegebenem Graphen $G(V, E)$ um einen Baum handelt.

2.5 Weitere Begriffe rund um Bäume

2.5.1 Vater und Sohn

Führt in einem Baum eine gerichtete Kante vom Knoten V zum Knoten S, so ist V der **Vater**, und S ein **Sohn**. Haben mehrere Knoten S_1, S_2, ... alle den gleichen Vater, so sind sie **Geschwister**.



Aufgabe 2.5.1

Welche Knoten in einem Baum sind nur Söhne aber keine Väter? Welche sind nur Vater aber nicht Sohn?

Aufgabe 2.5.2

Beweisen Sie: Ein Baum mit n Knoten hat $n - 1$ Kanten.

2.5.2 Grad eines Baums

Mit dem **Grad** eines Baums wird das Maximum der Ausgangsgrade aller Knoten angegeben, also die maximale "Verästelung" des Baums. Hat jeder Vater höchstens 2 Söhne, so hat der Baum Grad 2 und wird **binärer Baum** genannt. Beträgt der Grad eines Baums 3, so heisst er **ternärer Baum**. Allgemein heisst ein Baum mit Grad k ein k-tärer Baum.

Aufgabe 2.5.3

a) Wie viele Blätter kann ein Binärbaum der Höhe 5 maximal haben?
b) Wie viele sind es allgemein für eine Höhe h?

Aufgabe 2.5.4

Kehren wir zu unserem Beispiel auf Seite 7 zurück, in welchem eine Münze und danach ein Würfel geworfen wird. Welchen Grad und welche Höhe hat der Baum der möglichen Ausgänge?

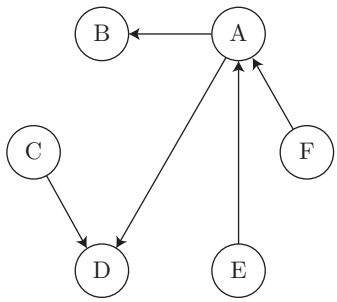
2.6 Zusammenfassung

Sie haben auf den vergangenen Seiten etliche neue Begriffe kennengelernt. Die folgende Übersicht soll Ihnen helfen, sich die wichtigsten wieder in Erinnerung zu rufen.

Wurzel	einziger Knoten eines Baums, zu dem keine Kante hinführt
Blatt	Baumknoten, von welchem keine kante weiterführt
innerer Knoten	Knoten eines Baums, der weder Wurzel noch Blatt ist
gerichteter Graph	Graph, dessen Kanten alle eine Richtung von Anfangs- zu Endknoten haben
Grad eines Knotens	(in ungerichteten Graphen) die Anzahl der Kantenansätze am Knoten
Eingangsgrad eines Knotens	(in gerichteten Graphen) die Anzahl der eingehenden Kanten
Ausgangsgrad eines Knotens	(in gerichteten Graphen) die Anzahl der ausgehenden Kanten
Weg / Pfad	Folge von Knoten, in der jeweils zwei aufeinander folgende Knoten mit Kanten verbunden sind
einfacher Weg	Weg, in welchem kein Knoten mehrfach vorkommt
Kreis	Weg, in welchem ausschliesslich Anfangs- und Endknoten gleich sind
Zyklus	Weg, in welchem Anfang- und Endknoten gleich sind. Dabei dürfen auch andere Knoten mehrfach vorkommen
zyklischer Graph	Graph, in welchem ein Zyklus existiert
azyklischer Graph	Graph, in welchem kein Zyklus existiert
Höhe eines Baums	Länge des längsten Pfades in einem Baum
Vater, Sohn	vom Vaterknoten führt eine Kante zum Sohnknoten
Geschwister	haben alle den gleichen Vaterknoten
Grad eines Baums	die maximale Anzahl von Söhnen eines Knotens
binärer Baum	Baum vom Grad 2

2.7 Lösungen der Aufgaben des Kapitels

Lösung zu Aufgabe 2.1.1



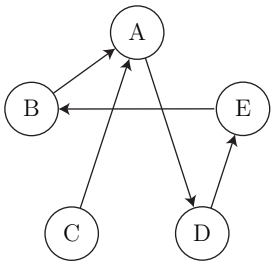
Lösung zu Aufgabe 2.1.2

$$
\begin{aligned}
V &= \{1,2,3,4,5,6\} \\
E &= \{(1,5),(2,3),(4,1),(4,6),(5,2),(6,6)\}
\end{aligned}
$$

Lösung zu Aufgabe 2.2.1

F hat sowohl den grössten Eingangsgrad (3) als auch den kleinsten Ausgangsgrad (0).

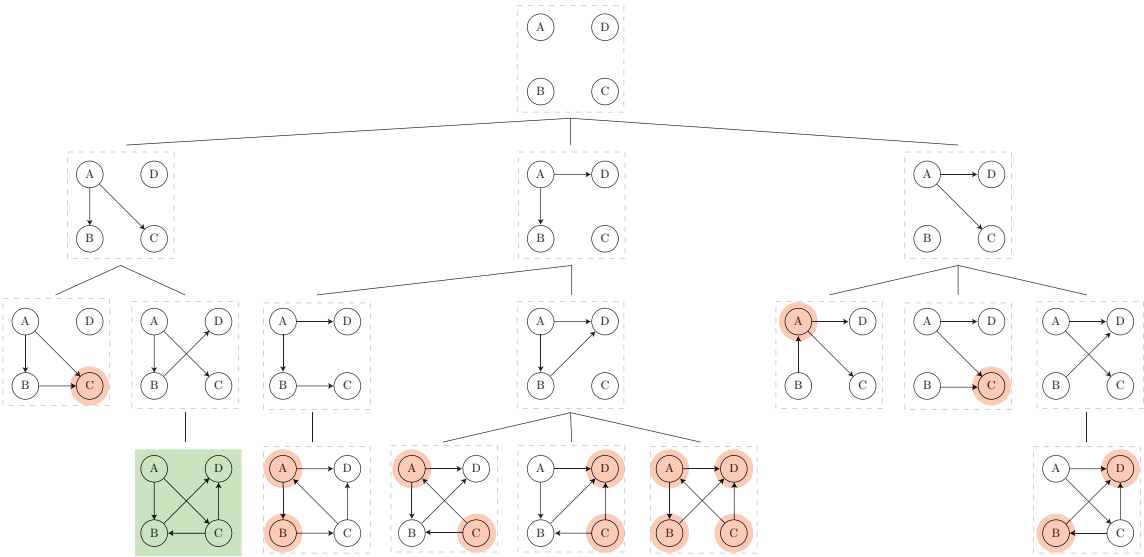
Lösung zu Aufgabe 2.2.2



Eingangsgrad(A) = 2, Ausgangsgrad(A) = 1;
Eingangsgrad(B) = 1, Ausgangsgrad(B) = 1;
Eingangsgrad(C) = 0, Ausgangsgrad(C) = 1;
Eingangsgrad(D) = 1, Ausgangsgrad(D) = 1;
Eingangsgrad(E) = 1, Ausgangsgrad(E) = 1;

Lösung zu Aufgabe 2.2.3

Um alle Lösungen zu finden, kann jede Möglichkeit durchprobiert werden, wie die Kanten zwischen den vier Knoten gelegt werden könnten. Die untenstehende Übersicht zeigt, dass es nur eine Lösung gibt:



Lösung zu Aufgabe 2.3.1

der kürzeste Weg ist (C,D,E,B).

Lösung zu Aufgabe 2.3.2

Einfache Wege von C nach A sind: (C,A) und (C,D,E,F,A)

Lösung zu Aufgabe 2.3.3

a) Es gibt nur einen Kreis: (A,E,C,H,D,A).
b) Der kürzeste Zyklus, der kein Kreis ist, ist (A,E,C,H,D,G,H,D,A).

Lösung zu Aufgabe 2.3.4

a) Es gibt 2 einfache Wege, nämlich (A,C,E,F), (A,C,G,F). Wege gibt es unendlich viele, da in C der Kreis (C,B,A,C) beliebig oft eingebaut werden kann. Beispiele sind (A,C,B,A,C,E,F) oder (A,C,B,A,C,B,A,C,E,F).
b) Kreise sind (A,C,B,A), (A,C,E,F,A) und (A,C,G,F,A).

Lösung zu Aufgabe 2.3.5

Sei k_1 der Anfangs- und k_n der Endknoten, für den es mehr Wege als einfache Wege gibt. Da sich einfache Wege und Wege nur dadurch unterscheiden, dass in letzteren Knoten mehrfach vorkommen können, muss es also einen Weg geben, in welchem dies auch geschieht. Sei k_i das erste Auftreten dieses Knotens im Weg und k_j das zweite, dann ist

$$Z = k_i, k_{i+1}, \ldots, k_j$$

ein Zyklus im Weg

$$k_1, \ldots, k_{i-1}, \underbrace{k_i, \ldots, k_j}_{Z}, k_{j+1} \ldots, k_n$$

Dieser Zyklus kann beliebig häufig durchlaufen werden, neben dem betrachteten Weg

$$k_1, \ldots, k_{i-1}, Z, k_{j+1} \ldots, k_n$$

sind also auch

$$\begin{gathered} k_1, \ldots, k_{i-1}, Z, Z, k_{j+1} \ldots, k_n \\ k_1, \ldots, k_{i-1}, Z, Z, Z, k_{j+1} \ldots, k_n \\ k_1, \ldots, k_{i-1}, Z, Z, Z, \ldots, k_{j+1} \ldots, k_n \end{gathered}$$

Wege, und dies sind unendlich viele. Damit ist die obige Behauptung bewiesen.

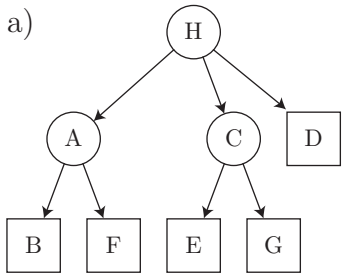
Lösung zu Aufgabe 2.4.1

a) ist zyklisch, (A,F,D,B,G,A) ist ein Zyklus. Die beiden anderen Graphen sind azyklisch.

Lösung zu Aufgabe 2.4.2

Lösungsweg für den Graphen a): Wir gehen durch alle Knoten und überprüfen den Eingangsgrad. Da nur Knoten H den Eingangsgrad 1 hat, ist die erste Bedingung der Definition eines Baumes erfüllt, und wir setzen Knoten H als Wurzel. Aus dem Graphen können wir nun entnehmen, dass von H aus drei Kanten ausgehen, nämlich zu A, C und D. Diese drei Knoten zeichnen wir in Tiefe 1 und ziehen die Kanten von der Wurzel zu diesen drei Knoten. Von A aus geht es weiter zu den Knoten B und F, die wir in Tiefe 2 zeichnen. Von C aus kommen wir zu E und G. D hat Ausgangsgrad 0, ist also selbst schon ein Blatt. Da auch B, F, E und G Ausgangsgrad 0 haben, sind wir fertig.

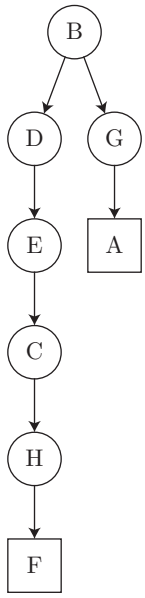
a)



Höhe $h = 2$

b) kein Baum

c)



Höhe $h = 5$

Lösung zu Aufgabe 2.4.3

Wir fügen jedem Knoten zwei Merkmale hinzu: jeder von ihnen kann auf "aktiv" oder auf "erledigt" gesetzt werden. Auf "aktiv" setzen wir alle Knoten, die wir von der Wurzel aus schon erreicht haben. Haben wir alle Kanten weiterverfolgt, die von einem aktiven Knoten ausgehen, so setzen wir ihn auf erledigt. Indem wir verlangen, dass von einem Knoten aus nur solche Knoten erreicht werden dürfen, die weder "aktiv" noch "erledigt" sind, stellen wir sicher, dass kein Zyklus vorhanden ist.

```
gehe über alle Knoten v_i ∈ V
   zähle alle v_i mit Eingangsgrad 0 → E_0
   zähle alle v_i mit Eingangsgrad 1 → E_1
falls (E_0 ≠ 1 ∨ E_1 ≠ |E| − 1)
   Ausgabe("Der␣Graph␣ist␣kein␣Baum")
sonst
   setze Knoten mit Eingangsgrad 0 auf 'aktiv'
   solange die Menge der aktiven Knoten nicht leer ist
      nehme einen aktiven Knoten v
      gehe über alle Knoten k_i, zu denen vom betrachteten, aktiven
         Knoten aus eine Kante weiterführt
```

```
        falls k_i schon 'aktiv' oder 'erledigt' ist
           Ausgabe("Der␣Graph␣ist␣kein␣Baum␣(es␣existiert␣ein␣
              Zyklus)")
           Stopp
        sonst
           markiere k_i als 'aktiv'
     markiere v als 'erledigt'
  Ausgabe("Der␣Graph␣ist␣ein␣Baum")
```

Lösung zu Aufgabe 2.5.1

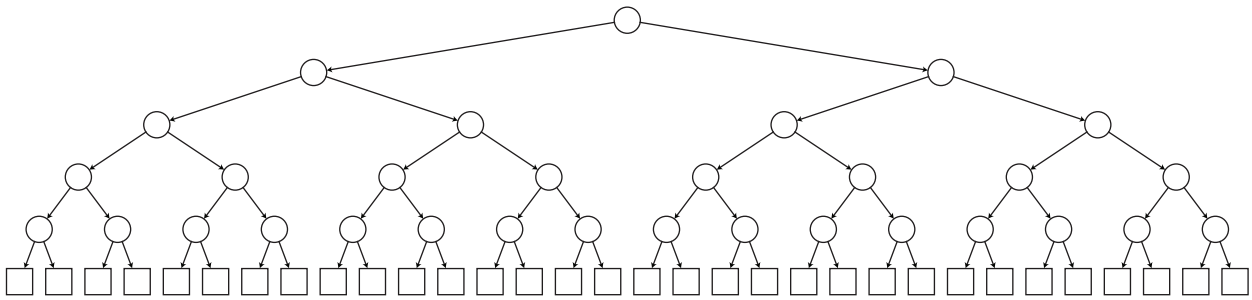
Nur Sohn: Blätter; nur Vater: Wurzel

Lösung zu Aufgabe 2.5.2

Mit Ausnahme der Wurzel, welche Eingangsgrad 0 hat, haben alle Knoten Eingangsgrad 1, das sind $n-1$ Knoten. Da jede Kante den Eingangsgrad genau eines Knotens um genau 1 erhöht, gibt es also auch $n-1$ Kanten.

Lösung zu Aufgabe 2.5.3

a) Die maximale Anzahl Blätter wird erreicht, wenn jeder innere Knoten und auch die Wurzel maximal viele Kinder, also zwei haben. Dann vervielfacht sich die Anzahl der Knoten pro Höhenschritt um den Faktor 2. Ein binärer Baum der Höhe 5 hat daher maximal $2^5 = 32$ Blätter.



b) Allgemein sind es 2^h Blätter.

Lösung zu Aufgabe 2.5.4

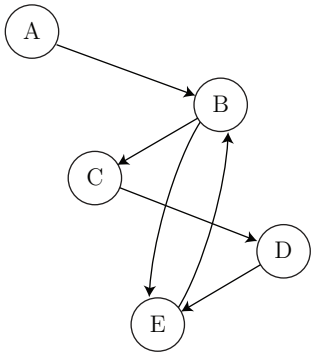
Er hat Grad 6 und eine Höhe von 2.

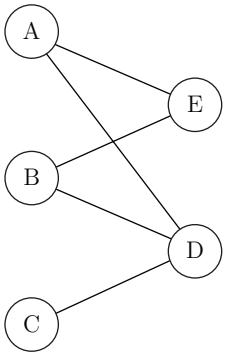
2.8 Kapiteltest I

Sie sind nun bereit für den ersten Kapiteltest.

Aufgabe 2.8.1

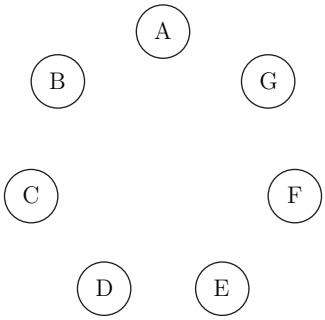
a) Welcher der beiden Graphen ist gerichtet, welcher ungerichtet?
b) Geben Sie für den gerichteten Graphen Ein- und Ausgangsgrad des Knotens E an.
c) Welche Graphen sind zyklisch? Geben Sie gegebenenfalls alle Kreis an.





Aufgabe 2.8.2

a) Die Knoten A,B,C,D,E,F,G des Graphen $G(V, E)$ sind schon gezeichnet. Es fehlen noch die gerichteten Kanten. $E = \{(\mathrm{C}, \mathrm{E}), (\mathrm{C}, \mathrm{G}), (\mathrm{D}, \mathrm{F}), (\mathrm{E}, \mathrm{D}), (\mathrm{E}, \mathrm{A}), (\mathrm{G}, \mathrm{B})\}$. Vervollständigen Sie die Zeichnung.



b) Ist der obige Graph ein Baum? Falls dem nicht so ist, begründen Sie Ihre Antwort. Andernfalls stellen Sie ihn als Baum dar.

Aufgabe 2.8.3

Ein Baum der Höhe 5 habe folgende Eigenschaften: die Wurzel hat zwei Söhne. Jeder innere Knoten hat 2 Söhne mehr als er Geschwister hat. Aus wie vielen Knoten besteht dieser Baum mindestens, aus wie vielen höchstens?

Kapitel 3

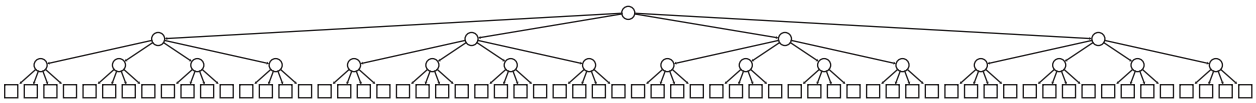
Das Generieren aller Möglichkeiten

Hier werden Sie lernen, wie Sie mit Hilfe eines geeigneten Baums alle Möglichkeiten generieren können, die gemäss Aufgabenstellung untersucht werden sollen.

3.1 Suchbaum - der Baum der Möglichkeiten

Sehen wir uns ein Sudoku für absolute Anfänger an. Es besteht aus 4×4 Kästchen, in welche die Zahlen 1 bis 4 so eingefüllt werden müssen, dass in jeder Zeile, in jeder Spalte und in jedem 2×2-Untertableau jede Zahl genau einmal vorkommt. Es wäre natürlich ein Leichtes, das Sukodu sofort zu vervollständigen. Hier wollen wir aber den Baum aller Möglichkeiten angeben, der entsteht, wenn in jedem leeren Feld einfach alles durchprobiert wird. Wir beginnen links oben, gehen nach rechts und dann in die nächste Zeile. In jeder Zelle setzen wir alle 4 Möglichkeiten ein. Damit hat der Suchbaum Grad 4 und Höhe 3, und seine 64 Blätter entsprechen allen Möglichkeiten, wie die drei offenen Felder mit einer Zahl von 1 bis 4 belegt werden kann.

1			2
3		1	4
4	1	2	3
2	3	4	1



In den Blättern stehen die Möglichkeiten als Tripel $(z_1|z_2|z_3)$. Das Tripel (4|3|2) wäre die Lösung des obigen Sudokus, aber so weit sind wir noch nicht. Alle Blätter des Baums bilden zusammen den Suchraum, also die Menge sämtlicher Möglichkeiten. Die inneren Knoten korrespondieren unvollständigen Möglichkeiten, die wir Teilmöglichkeiten nennen werden. Ein Knoten in Tiefe 1 beispielsweise gehört zu einer Teilmöglichkeit, in welcher das erste Feld schon durch eine Ziffer belegt worden ist, die restlichen aber noch nicht.

Die Lösung unseres Anfänger-Sudoku-Problems ist eine Teilmenge des Suchraums. Das Sudoku-Problem ist daher ein Suchproblem, in welchem es darum geht,

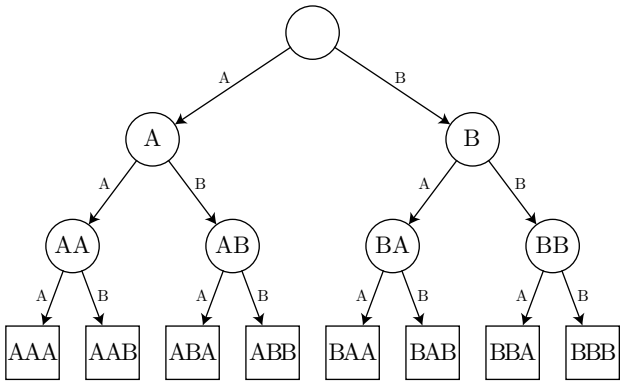
- den Suchraum über einen Baum zu beschreiben und
- in diesem Baum die Lösung(en) zu finden.

Der Suchraum ist die Menge aller für ein Problem bestehenden Lösungskandidaten.

Der Suchbaum erzeugt den Suchraum und ermöglicht eine Ordnung.

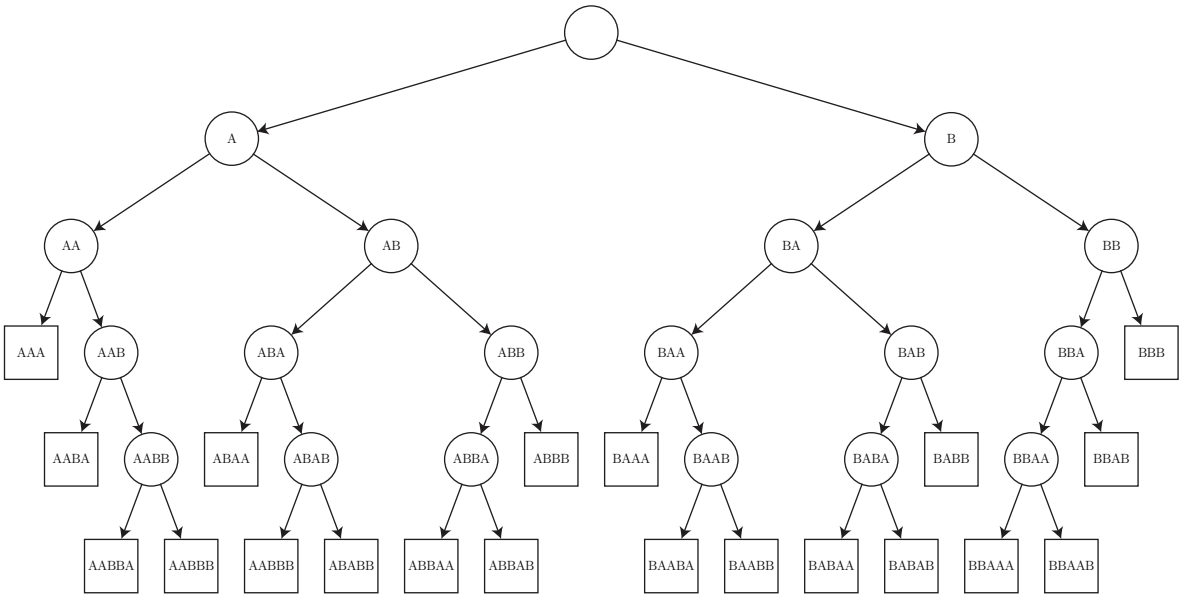
Beispiel 1

Hier sei ein weiteres Beispiel angeführt, wie der Suchraum durch einen Baum erzeugt werden kann. Aus den beiden Buchstaben A, B soll dreimal nacheinander einer gewählt werden. Der zugehörige Baum aller Möglichkeiten (die in den Blättern stehen) sieht folgendermassen aus:



Beispiel 2 - Der Suchbaum für einen Tennis-Match auf 3 Sätze

Betrachten wir einen Tennis-Match zweier Spieler A und B, in welchem derjenige gewinnt, der zuerst drei Siegessätze vorweisen kann. Im Baum geben wir nun nach jedem Satz an, wie der momentane Spielstand ist, eine Angabe wie "ABB" würde also beispielsweise bedeuten, dass Spieler A den ersten, Spieler B dann die beiden folgenden Sätze gewonnen hat. Der komplette Baum ist:



Alle hier auftretenden Blätter sind mögliche Ausgänge des Spiels. Es ist also durchaus möglich, dass nicht nur die Blätter der untersten Stufe als gültige Möglichkeiten angesehen werden müssen.

Machen Sie sich in den folgenden Aufgaben mit der Erzeugung und den Eigenschaften solcher Suchbäume vertraut.

Aufgabe 3.1.1

Aus den drei Zahlen 1, 2 und 3 kann drei Mal eine gewählt werden, wobei die gleiche Zahl auch mehrfach vorkommen darf.

a) Zeichnen Sie den zugehörigen Baum aller Möglichkeiten.
b) Zeichnen Sie den Suchbaum, wenn nur Tripel erlaubt sind, in denen jede nachfolgende Zahl mindestens so gross wie ihr Vorgänger ist.

Aufgabe 3.1.2

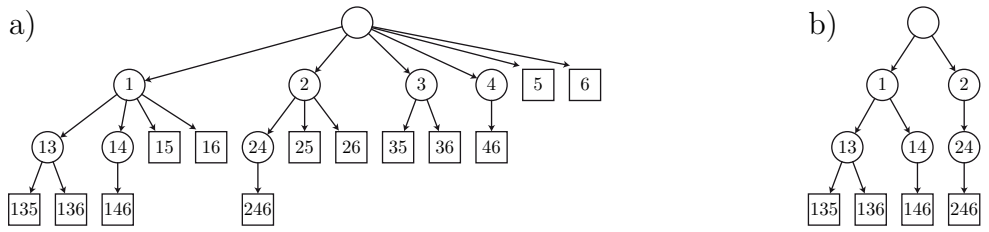
Aus den Zahlen 1 bis 4 kann wiederholt gezogen werden. Dabei soll die Summe aller gezogener Zahlen genau 5 betragen. Zeichnen Sie den entsprechenden Baum aller Möglichkeiten.

Aufgabe 3.1.3

Im Folgenden sehen Sie zwei verschiedene Möglichkeitenbäume zur gleichen Aufgabe. Beide sind richtig, im Baum a) stehen die vollständigen Möglichkeiten nur in den Blättern der untersten Reihe. Die Aufgabenstellung hiess:

"Zeichnen Sie einen Suchbaum, der alle Möglichkeiten generiert, wie aus der Zahlenmenge $\mathbb{M} = \{1, 2, 3, 4, 5, 6\}$ Tripel (i, j, k) zusammengesetzt werden können, so dass jede Zahl um mindestens 2 grösser als ihr Vorgänger ist, d.h $j \geq i + 2, k \geq j + 2$."

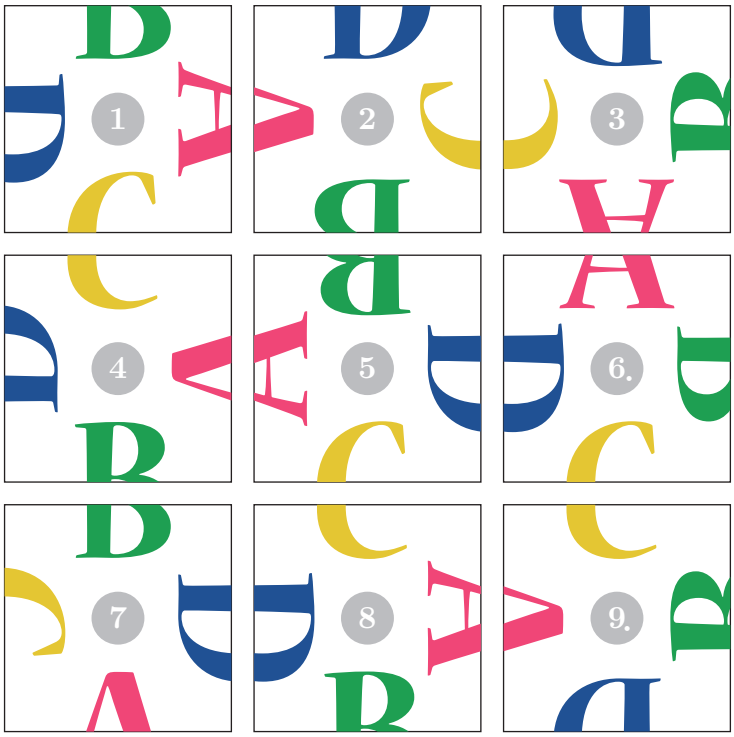
Der Unterschied kommt von unterschiedlichen Algorithmen, die zur Erzeugung der Bäume verwendet wurden. Geben Sie in Worten zu beiden Bäumen einen Algorithmus an, wie die Söhne eines inneren Knotens erzeugt wurden.



Der Suchbaum kann unterschiedlich effizient aufgebaut werden. Ist der Suchraum sehr mächtig, müssen also sehr viele Möglichkeiten generiert werden, so werden wir bemüht sein, einen möglichst effizienten Algorithmus anzugeben, der keine Äste enthält, die zu keiner gültigen Möglichkeit führen.

Aufgabe 3.1.4 - Das verflixte Puzzlespiel

Mit den folgenden 9 Karten soll ein Quadrat so gebildet werden, dass alle aneinander grenzenden Kanten einen Buchstaben bilden. Im abgebildeten Fall ist dies fast erfüllt, nur die Kante zwischen Karte 3 und Karte 6 zeigt zweimal der oberen Teil eines Grossbuchstabens A. (Die in den grauen Kreisen angegebenen Nummern dienen nur dazu, die Karten hier einfacher beschreiben zu können.) Durch Umlegen und Drehen soll nun erreicht werden, dass alle Kanten zueinander passen.



Beschreiben Sie, wie der Suchbaum für diese Aufgabe angeordnet werden kann, wenn einfach jede Möglichkeit durchprobiert werden soll. Wie viele Möglichkeiten ergeben sich? Wie können symmetrische Lösungen zusammengefasst werden? (Es sind alle Karten paarweise verschieden.)

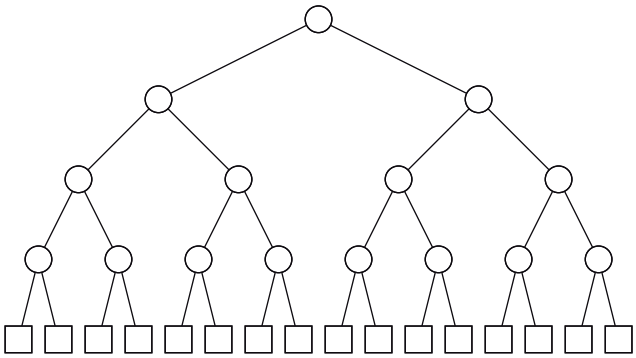
3.2 Bäume zeichnen

Als Vorbereitung für die Programmierung von Suchbäumen schreiben wir ein Programm, das uns einen Baum zeichnet.

Mit Hilfe eines Programms soll ein binärer Baum von (theoretisch) beliebiger Höhe gezeichnet werden. Die erste Zeile des Programms lautet daher:

```
int h=4;    // Höhe des Baums
```

Der erzeugte Baum soll so aussehen:



Natürlich kann der Baum auf verschiedene Arten programmiert werden. Sehen wir uns hier an, wie er rekursiv erzeugt werden kann. Der Baum ist grösstenteils aus einem "Bausteinen" aufgebaut, der aus einem Knoten und zwei ausgehenden Kanten besteht. Damit sich die Geschwisterknoten nicht gegenseitig überlappen, muss die Breite eines solchen Bausteins halbiert werden, wenn eine Stufe in die Tiefe gegangen wird. Zu unterst angekommen wird statt eines weiteren Bausteins ein Blatt angehängt.

Das Programm (im Pseudocode) kann also folgendermassen rekursiv arbeiten:

```
zeichneVerzweigung(x, y, breite, hoehe, verbleibendeStufen)
   falls (verbleibendeStufen>0)
      zeichne Kanten und Knoten
      zeichneVerzweigung(x-breite/2,y+hoehe, breite/2, hoehe,
         verbleibendeStufen-1)
      zeichneVerzweigung(x+breite/2,y+hoehe, breite/2, hoehe,
         verbleibendeStufen-1)
   sonst
      zeichne Blatt
```

Aufgabe 3.2.1

Schreiben Sie ein Programm, welches einen binären Baum mit h Stufen zeichnet.

3.3 Suchbäume programmieren

Aufgabe 3.3.1- zur Einführung

a) Schreiben Sie ein Programm, das Ihnen alle vierstelligen Zahlen ausgibt, die mit den Ziffern 1,2,3 gebildet werden können.
b) Wie kann das Programm erweitert werden, dass es ohne grosse Änderung für n-stellige Zahlen (mit den gleichen Ziffern 1,2,3) funktioniert?

Die obige Aufgabe kann auf verschiedene Arten gelöst werden. Wir sehen uns zwei an:

1. Ansatz: verwendet vier ineinander geschachtelte Schlaufen

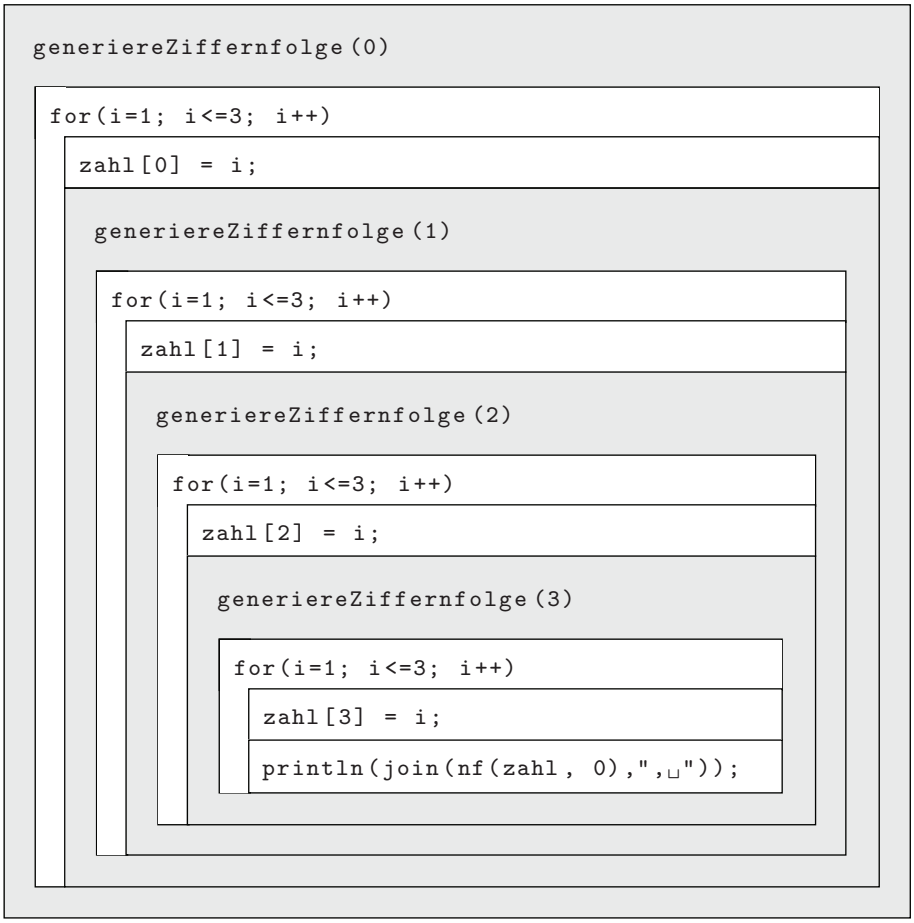
```
int[] zahl = new int[4];
int i, j, k, l;

for (i=1; i<=3; i++)
{
  zahl[0]= i;
  for (j=1; j<=3; j++)
  {
    zahl[1] = j;
    for (k=1; k<=3; k++)
    {
      zahl[2] = k;
      for (l=1; l<=3; l++)
      {
        zahl[3] = l;
        println(join(nf(zahl, 0), ", "));  // Ausgabe des Arrays
      }
    }
  }
}
```

Dieser Code ist für Teilaufgabe b) jedoch ungeeignet, da für n Ziffern auch n Schlaufen programmiert werden müssen. Ist n variabel, so ist funktioniert dieser Ansatz nicht.

2. Ansatz: rekursiv

Die vier ineinander geschachtelten Schlaufen können auch erreicht werden, indem sich eine Funktion, welche eine Schlaufe enthält, in der Schlaufenabarbeitung rekursiv aufruft.

Auf der nächsten Seite finden Sie ein entsprechendes Struktogramm. Wir beginnen mit `generiereZiffernfolge(0)`. Dies führt zu einem ersten Funktionsaufruf, in welchem die Variable i definiert wird (lokal) und dann ein Schlaufe von $i = 1$ bis $i = 3$ durchgeführt wird. In der Schlaufe wird zuerst $i = 1$ in $zahl[0]$ geschrieben, dann wird `generiereZiffernfolge(1)` aufgerufen, und so fort:



Der entsprechende Code für Länge 4 (Teilaufgabe a) ist

```
int[] zahl = new int[4]; // global definiert

void setup()
{
  generiereZiffernfolge(0);
}

void generiereZiffernfolge(int momTiefe)
{
  int i;
  for(i=1; i<=3; i++)  // Ziffern von 1 bis 3
  {
    zahl[momTiefe] = i;
    if (momTiefe==3)  // sind bei zahl[3], also der vierten Stelle
       angelangt
      println(join(nf(zahl, 0), ", "));  // Ausgabe des Arrays
    else
      generiereZiffernfolge(momTiefe+1);
  }
}
```

Ohne Probleme lässt er sich für beliebiges n erweitern.

```
static int N=4; // Anzahl Ziffern
int[] zahl = new int[N]; // global definiert

void setup()
{
  generiereZiffernfolge(0);
}

void generiereZiffernfolge(int momTiefe)
{
  int i;
  for(i=1; i<=3; i++)  // Ziffern von 1 bis 3
  {
    zahl[momTiefe] = i;
    if (momTiefe==N-1)  // sind bei zahl[N-1], also der N-ten Stelle
        angelangt
      println(join(nf(zahl, 0), ", "));  // Ausgabe des Arrays
    else
      generiereZiffernfolge(momTiefe+1);
  }
}
```

Die rekursive Programmierung eignet sich gut, um eine Baumstruktur des Suchraums aufzubauen, da sie erlaubt, nur die Anweisungen für die Generierung der Söhne von Knoten auf einer Tiefe `momTiefe` anzugeben. Es ist jedoch durchaus möglich, den Suchbaum auch iterativ (statt rekursiv) aufzubauen. Wie dies geht, können Sie bei Interesse im Kapitel 3.4 nachlesen.

Aufgabe 3.3.2

a) Es sollen alle dreistelligen Zahlen mit den Ziffern 0,1 ausgegeben werden, wobei auch führende Nullen erlaubt sind. Zeichnen Sie den zugehörigen Baum.

b) Schreiben Sie ein Programm, das Ihnen alle n-stellige Zahlen mit den Ziffern 0,1 ausgibt. Testen Sie das Programm für $n = 3$ und $n = 4$ und zählen Sie die Anzahl der ausgegebenen Möglichkeiten.

c) Wie viele Möglichkeiten erwarten Sie für $n = 10$ und $n = 50$?

Aufgabe 3.3.3

a) Programmieren Sie mit Hilfe eines Baums die Ausgabe aller n-stelliger Zahlen mit Ziffern $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, wobei jede Ziffer ab der zweitvordersten Stelle grösser als ihr Vorgänger (Ziffer an der nächst höheren Stelle) sein muss.

b) Bestimmen Sie für jedes n die Anzahl solcher Zahlen, wenn die Zahl eine führende Null aufweisen darf.

c) Wie viele solche Zahlen gibt es, wenn keine führende Null erlaubt ist? Geben Sie die Anzahlen wiederum für jedes n an.

Aufgabe 3.3.4

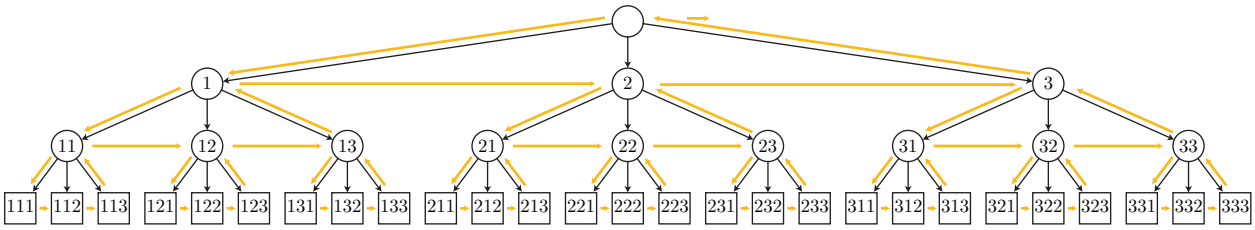
a) Schreiben Sie ein Programm, dass alle Zahlen (im Dezimalsystem) mit höchstens 6 Stellen zählt, deren einfache Quersumme 8 beträgt.
b) Begründen Sie die gefundenen Werte mit kombinatorischen Betrachtungen.

Die einfache Quersumme ist die Summe aller Ziffern. (Im Gegensatz dazu wird für die einstellige Quersumme die Quersumme der Quersumme solange genommen, bis sie nur noch eine Stelle hat. Beispiel: Die einfache Quersumme der Zahl 263747 ist 29, die einstellige Quersumme aber 2, da die einfache Quersumme von 263747 eben 29 ist, die Quersumme von 29 dann 11 und von 11 dann 2.)

Das folgende Unterkapitel muss nicht zwingend behandelt werden. Es beschreibt, wie ein Suchbaum iterativ statt rekursiv aufgebaut wird, was zu einem Zeit- und Speicherplatzgewinn führen kann. Die Betrachtungen im nächsten Kapitel werden aber alle rekursiv programmiert werden. Vielleicht interessiert es Sie aber dennoch, wie ein Baum iterativ generiert werden kann.

3.4 Iterative Implementierung des Suchbaums*

Die rekursive Programmierung ist nicht zwingend, der Baum kann immer auch iterativ durchsucht werden. Zur Veranschaulichung nehmen wir das Beispiel der n-stelligen Zahl mit den Ziffern $1, 2, 3$, das Sie aus dem vergangenen Unterkapitel schon kennen. Hier ist es allerdings bequemer, die Ziffern in `zahl[1]` bis `zahl[n]` zu speichern (statt von `zahl[0]` bis `zahl[n-1]` in der rekursiven Programmierung). Für $n = 3$ werden wir die Knoten des Suchbaums in der eingezeichneten Reihenfolge generieren:



Der iterative Ablauf geht so: Wir beginnen ganz links im Baum, also mit `zahl[1]` = `zahl[2]` = ... = `zahl[n]` = 1. Zusätzlich brauchen wir eine Variable, welche die Stufe anzeigt, auf welcher wir uns befinden. Zu Beginn sind wir auf der untersten Stufe $i = n$ im vordersten Blatt. Nun geben wir die entsprechende Zahl aus und wiederholen das Folgende: Wir erhöhen die Ziffer auf der betrachteten Stufe um 1. Wird sie dadurch aber 4, so entspricht dies keinem Geschwisterblatt mehr, und wir klettern den Baum so weit hinauf, bis wir eine Ziffer erhöhen können, ohne dass sie dadurch 4 wird. Vom neuen Geschwisterknoten aus nehmen wir dann wiederholt so lange den ersten Sohn, bis wir in einem Blatt des Baums angekommen sind, welches die nächste berechnete Möglichkeit ist.

Wir sind mit der Generierung des Baums fertig, wenn wir so hoch klettern mussten, dass `zahl[1]` auf 4 und daher die Wurzel `zahl[0]` um 1 erhöht werden sollte.

Das iterative Programm sieht also so aus:

```
static int N = 3;
int i, j;
int[] zahl = new int[N+1];

void setup()
{
  for(i=0; i<= N; i++)
    zahl[i]=1;
  while(zahl[0]==1)  // Wurzel noch unverändert
  {
    i = N; // unterste Stufe im Baum, also hinterste Stelle der Zahl
    print("Zahl:␣");  // Ausgabe
    for(j=1; j<=N; j++)
      print(zahl[j]);  // gebe Möglichkeit aus
    print('\n');
    zahl[i]++;  // gehe zum nächsten Geschwister
    while(zahl[i]==4)  // falls Geschwister nicht existiert
    {
      i--;  // gehe zum Vater
      zahl[i]++;  // nächstes Geschwister
    }
    while(i<N)  // wieder die jeweils ersten Söhne nehmen
    {
      i++;
      zahl[i]=1;
    }
  }
}
```

Aufgabe 3.4.1

Ändern Sie das oben angegebene Programm so ab, dass es alle 5-stelligen Dualzahlen ausgibt.

Aufgabe 3.4.2

Schreiben Sie ein Programm, das alle 3-stelligen Zahlen im Dezimalsystem ausgibt, deren Ziffern von links nach rechts gelesen grösser werden. In der Zahl $z = z_2 \cdot 10^2 + z_1 \cdot 10 + z_0$ muss also $z_0 > z_1$ und $z_1 > z_2$ sein.

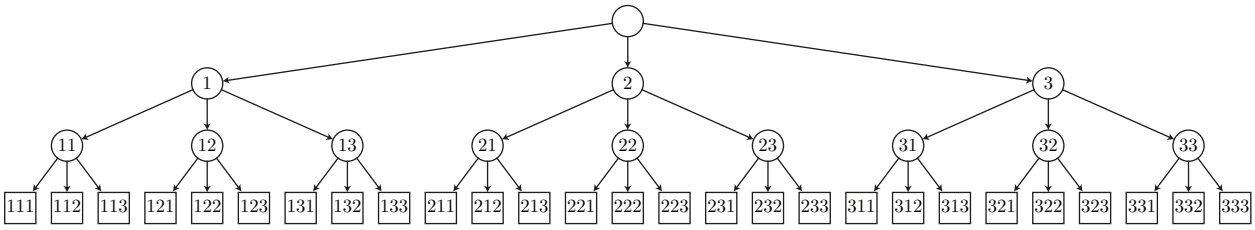
3.5 Zusammenfassung

- In diesem Kapitel haben Sie gelernt, wie ein Suchraum mit Hilfe eines Baums aufgespannt werden kann, und Sie können einen solchen Suchbaum zeichnen.
- Sie können einen Suchbaum rekursiv programmieren.
- Wenn Sie das letzte Unterkapitel bearbeitet haben, kennen Sie auch eine Möglichkeit, den Baum iterativ zu generieren.
- Sie haben gesehen, dass ein Suchbaum verschieden effizient aufgebaut werden kann, und sollten sich daher jeweils überlegen, welches für eine gegebene Aufgabe die effizienteste Methode darstellt.

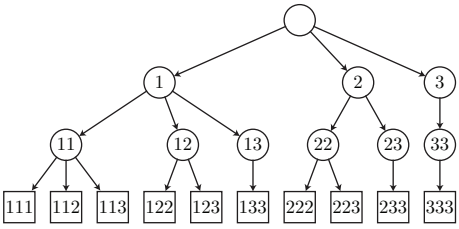
3.6 Lösungen der Aufgaben des Kapitels

Lösung zu Aufgabe 3.1.1

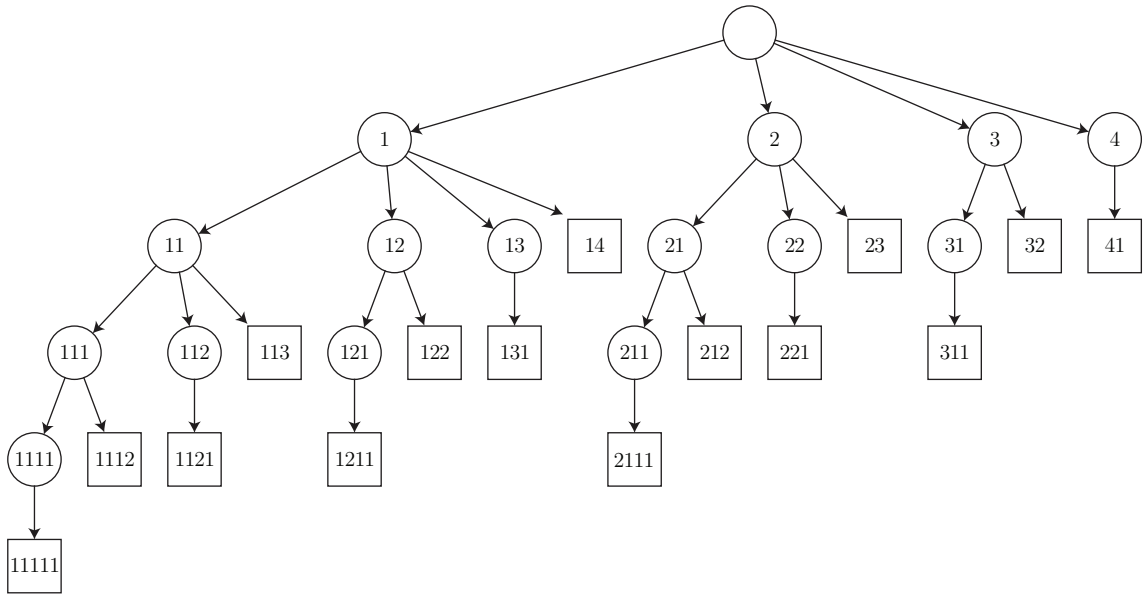
a)



b)



Lösung zu Aufgabe 3.1.2



Lösung zu Aufgabe 3.1.3

Im Algorithmus zum Baum a) wurden in jedem Knoten folgendermassen vorgegangen: ist m die zuletzt angefügte Zahl, so wurde die bestehende Teilmöglichkeit jeweils mit einer Zahl aus $[m+2, 6]$ erweitert. Die Möglichkeiten des Problems sind dann nur die Blätter mit der Tiefe 3, also nur die "untersten".

Im Algorithmus für Baum b) wurde effizienter vorgegangen, indem in den Knoten der 1. Stufe höchstens bis zur Ziffer 2 zugelassen wurde und in der 2. Stufe höchstens bis zur Ziffer 4, denn höhere Ziffern können zu keiner gültigen Möglichkeit führen. Allgemein ist das Verfahren für einen Knoten der Tiefe t das folgende: ist wiederum m die zuletzt angefügte Zahl, dann wird die Zwischenlösung jeweils mit einer Zahl aus $[m+2, 6-2(3-t)] = [m+2, 2t]$ erweitert. ($(3-t)$ entspricht nämlich der Anzahl fehlender Zahlen bis zur vollständigen Lösung.) Dadurch entstehen keine unnötigen Äste.

Lösung zu Aufgabe 3.1.4

Wir stellen uns vor, dass wir zuerst die linke obere Ecke des noch leeren Quadrats mit einer Karte belegen. Dazu haben wir 9 Möglichkeiten. Für das horizontal daneben liegende Feld haben wir dann noch 8 Möglichkeiten. Fahren wir fort, so ergeben sich 9! Möglichkeiten, wie die Karten im Quadrat verteilt werden können. Jede Karte kann nun auch noch in 4 Richtungen (jeweils 90° gedreht) liegen. Daraus folgen $9! \cdot 4^9$ mögliche Anordnungen. Dabei entsprechen sich immer 4 Möglichkeiten, die durch Drehung des ganzen Quadrats um Vielfache von 90° auseinander hervorgehen. Um jeweils nur einen Repräsentanten zu generieren, geben wir die Ausrichtung einer Karte (beispielsweise der Karte 1) vor - sie soll nicht gedreht werden dürfen. Dann haben wir $9! \cdot 4^8 = 2.38 \cdot 10^{10}$ Möglichkeiten.

Lösung zu Aufgabe 3.2.1

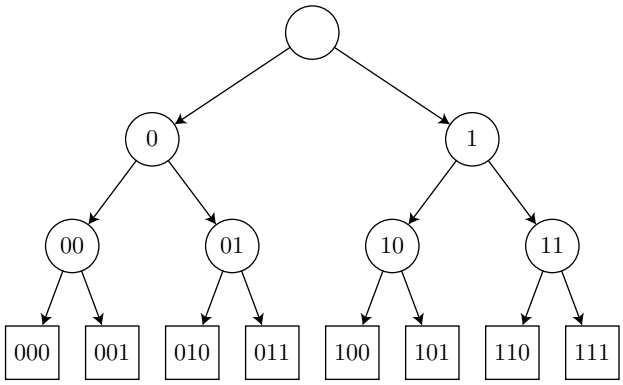
```
int h=4;    // Höhe des Baums

void setup()
{
  size(1000,500);
  noLoop();
}

void draw()
{
  background(255);
  zeichneVerzweigung(500,20,pow(2,h)*15,60,h);
}


void zeichneVerzweigung(float x,float y,float breite, float hoehe,
   int verbleibendeStufen)
{
  if(verbleibendeStufen>0)
  {
    line(x, y, x-breite/2, y+hoehe);
    line(x, y, x+breite/2, y+hoehe);
    ellipse(x,y,20,20);
    zeichneVerzweigung(x-breite/2, y+hoehe, breite/2, hoehe,
       verbleibendeStufen-1);
    zeichneVerzweigung(x+breite/2, y+hoehe, breite/2, hoehe,
       verbleibendeStufen-1);
  }
  else
    rect(x-10,y-10,20,20);
}
```

Lösung zu Aufgabe 3.3.1

Die Lösung ist im Text erklärt.

Lösung zu Aufgabe 3.3.2

a)



b)

```
static int N=3; // Anzahl Ziffern
int[] zahl = new int[N]; // global definiert

void setup()
{
  generiereZiffernfolge(0);
}

void generiereZiffernfolge(int momTiefe)
{
  int i;
  for(i=0; i<=1; i++)  // Ziffern von 0 bis 1
  {
    zahl[momTiefe] = i;
    if (momTiefe==N-1)  // sind bei zahl[N-1], also der N-ten
       Stelle angelangt
      println(join(nf(zahl, 0), ", "));  // Ausgabe des Arrays
    else
      generiereZiffernfolge(momTiefe+1);
  }
}
```

Das Programm ergibt alle n-stelligen Binärzahlen, also 2^n Möglichkeiten. Für $n = 3$ sind dies 8, für $n = 4$ dann 16.

c) Für $n = 10$ ergeben sich $2^{10} = 1\,024$ und für $n = 50$ sogar $1.13 \cdot 10^{15}$ Möglichkeiten.

Lösung zu Aufgabe 3.3.3

a)

```
static int N=4; // Anzahl Ziffern
int[] zahl = new int[N]; // global definiert
int anzahl=0;

void setup()
{
  generiereZiffernfolge(0);
  println("Es␣gibt␣"+anzahl+"␣Lösungen");
}

void generiereZiffernfolge(int momTiefe)
{
  int i;
  int s; // Schranke der Schlaufe

  if(momTiefe == 0)  // vorderste Stelle
    s = 0;   // für Teilaufgabe c) s = 1
  else
    s = zahl[momTiefe-1]+1;
  for(i=s; i<=9; i++)
  {
    zahl[momTiefe] = i;
    if (momTiefe==N-1)  // hinterste Stelle erreicht
    {
      println(join(nf(zahl, 0), ",␣")); // Ausgabe des Arrays
      anzahl++;
    }
    else
      generiereZiffernfolge(momTiefe+1);
   }
}
```

b) Mit führender Null kann es höchstens 10-stellige Zahlen geben, wobei die einzige 10-stellige Zahl die Zahl 0123456789 ist.

n	Anzahl Möglichkeiten
1	10
2	45
3	120
4	210
5	252
6	210
7	120
8	45
9	10
10	1

Es sind dies die Binomialkoeffzienten $\binom{10}{n}$, denn das Problem ist ein Ziehen ohne Zurücklegen und ohne Reihenfolge der n gebrauchten Ziffern aus den 10 vorhandenen. Die Anordnung der Ziffern ist danach eindeutig, da sie in aufsteigender Grösse (von links nach rechts gesehen) geschrieben werden müssen.

c) Kann die Null nicht zuvorderst stehen, so ist sie nirgends in der Zahl erlaubt. Also ist das Problem äquivalent zu dem in Teilaufgabe a) wobei nur die Ziffern $1, 2, \ldots, 9$ verwendet werden. Die Anzahl Möglichkeiten für n-stellige Zahlen ist dann $\binom{9}{n}$.

Lösung zu Aufgabe 3.3.4

a)

```
static int N = 6;
static int QUERSUMME = 8;
int zaehler = 0;
int[] zahl = new int[N]; // global

void setup()
{
  generiereZiffernfolge(0,0);
  println("Es␣gibt␣"+zaehler+"␣höchstens␣"+N+"-stellige␣Zahlen␣
     mit␣Quersumme␣"+QUERSUMME);
}

void generiereZiffernfolge(int momTiefe, int momQuersumme)
{
  int i, imin, qsDiff;

  if(momTiefe == N-1)  // hinterste Stelle, die geforderte
     Quersumme muss gebildet werden
  {
    qsDiff = QUERSUMME - momQuersumme;
    if(qsDiff <= 9)   // diese Abfrage ist nur für QUERSUMME >
       9 nötig, schadet aber nicht
    {
      zahl[momTiefe] = qsDiff;
      // println(join(nf(zahl, 0), ", "));  // falls Zahlen
         ausgegeben werden sollen
      zaehler++;
    }
  }
  else
  {
    qsDiff = QUERSUMME - momQuersumme;
    qsDiff = min(qsDiff, 9); // höchstens Ziffer 9 zulassen;
    for(i=0; i<=qsDiff; i++)
    {
      zahl[momTiefe] = i;
      generiereZiffernfolge(momTiefe+1, momQuersumme+i);
    }
  }
}
```

Es gibt 1 287 höchstens 6-stellige Zahlen mit Quersumme 8.

b) Es handelt sich hier (da die einfache Quersumme ≤ 9 ist) um ein einfaches Ziehen mit Zurücklegen ohne Reihenfolge. Es wird $k = 8$ Mal eine Stelle aus den $n = 6$ zur Verfügung stehenden gezogen und dieser dann Eins gutgeschrieben. Also gibt es

$$\binom{n+k-1}{k} = \binom{13}{8} = 1\,287 \text{ Möglichkeiten}$$

Lösung zu Aufgabe 3.4.1

Die Ziffern gehen nun von 0 bis 1. Zum Vater wird gewechselt, wenn `zahl[i]==2` ist.

```
static int N = 5;
int i, j;
int[] zahl = new int[N+1];

void setup()
{
  for(i=0; i<= N; i++)
    zahl[i]=0;
  while(zahl[0]==0)  // Wurzel noch unverändert
  {
    i = N; // unterste Stufe im Baum, also hinterste Stelle der Zahl
    print("Zahl:␣");  // Ausgabe
    for(j=1; j<=N; j++)
      print(zahl[j]);  // gebe Möglichkeit aus
    print('\n');
    zahl[i]++;  // gehe zum nächsten Geschwister
    while(zahl[i]==2)  // falls Geschwister nicht existiert
    {
      i--;  // gehe zum Vater
      zahl[i]++;  // nächstes Geschwister
    }
    while(i<N)  // wieder die jeweils ersten Söhne nehmen
    {
      i++;
      zahl[i]=0;
    }
  }
}
```

Lösung zu Aufgabe 3.4.2

```
int N = 3;  // Anzahl Stellen
int i, j;
int[] zahl = new int[N+1];
int anzahlEinsen = 0;

void setup()
{
  for(i=0; i<= N; i++)
    zahl[i]=i;
  while(zahl[0]==0)  // Wurzel noch unverändert
  {
    i = N; // unterste Stufe im Baum, also hinterste Stelle der Zahl
    if(zahl[i]<=9)
    {
      print("Zahl:␣");  // Ausgabe
      for(j=1; j<=N; j++)
        print(zahl[j]);  // gebe Möglichkeit aus
      print('\n');
    }
    zahl[i]++;
    while(zahl[i]>9)  // falls Geschwister nicht existiert
    {
      i--;  // gehe zum Vater
      zahl[i]++;
    }
    while(i<N)  // wieder die jeweils ersten Söhne nehmen
    {
      i++;
      zahl[i]=zahl[i-1]+1;
    }
  }
}
```

3.7 Kapiteltest II

Prüfen Sie Ihre Fähigkeiten mit dem folgenden Kapiteltest.

Aufgabe 3.7.1

Zeichnen Sie einen Baum, der alle Möglichkeiten aufspannt, wie die drei Symbole ■▲● aneinander gereiht werden können.

Aufgabe 3.7.2

a) Schreiben Sie ein Programm, das Ihnen alle Möglichkeiten ausgibt, wie die Zahlen 1 bis n verschieden angeordnet werden können.
b) Wie viele Möglichkeiten gibt es bei n Zahlen?

Aufgabe 3.7.3

Gesucht sind alle n-stellige Zahlen der Form $c_n \cdot 10^n + c_{n-1} \cdot 10^{n-1} + \ldots + c_1 \cdot 10 + c_0$ für die $c_{i+1} > c_i$ $(0 \leq i < n)$ gilt und deren Zuwachs der Ziffer von Stelle zur nächstkleineren Stelle konstant ist. Beispiele solcher dreistelliger Zahlen sind 123 oder 357, ein Beispiel einer solchen 4-stelliger Zahl ist 1357.

a) Zeichnen Sie für dreistellige Zahlen ($n = 3$) einen effizienten Suchbaum, der keine überflüssigen Äste enthält.
b) Schreiben Sie ein Programm, das für $n \geq 3$ alle Möglichkeiten ausgibt. Vergleichen Sie das Resultat für $n = 3$ mit Teilaufgabe a).

Aufgabe 3.7.4

Erstellen Sie ein Programm, das alle fünfstelligen Dezimalzahlen zählt, die (ohne Rest) durch 3 teilbar sind und aussschliesslich aus ungeraden Ziffern bestehen.

Kapitel 4

Backtracking

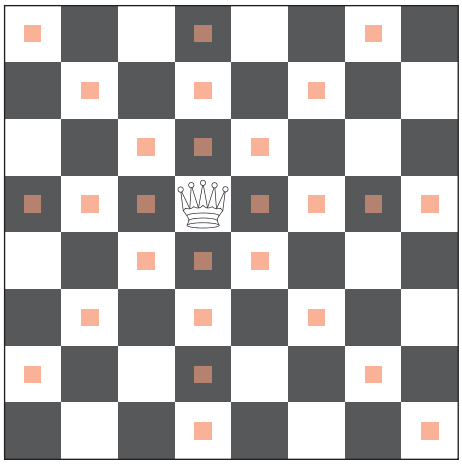
Häufig kann ein Baum nicht direkt so konstruiert werden, dass nur Lösungen entstehen. Wir generieren dann einen grösseren Baum aller in Frage kommenden Möglichkeiten und testen diese, ob sie auch wirklich Lösungen sind.

Bemerken wir beim Generieren einer Teilmöglichkeiten, dass diese unter keinen Umständen zu einer vollen Lösung ausgebaut werden kann, verfolgen wir den entsprechenden Ast im Baum auch nicht mehr weiter.

4.1 Einfaches Backtracking

4.1.1 Einführung anhand des Damenproblems

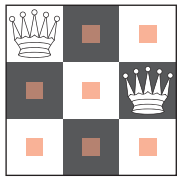
1848 stellte der bayrische Schachmeister Max Bezzel in einer Schachzeitung das folgende schachmathematische Problem:
"Wie viele Möglichkeiten gibt es, 8 Damen auf einem Schachbrett so aufzustellen, dass keine Dame eine andere bedroht?"
Eine Dame bedroht dabei jede Figur, die sie auf horizontalem, vertikalem oder diagonalem Weg direkt erreichen kann. In der nebenstehenden Figur sind diese Felder mit einem roten Quadrat gekennzeichnet.



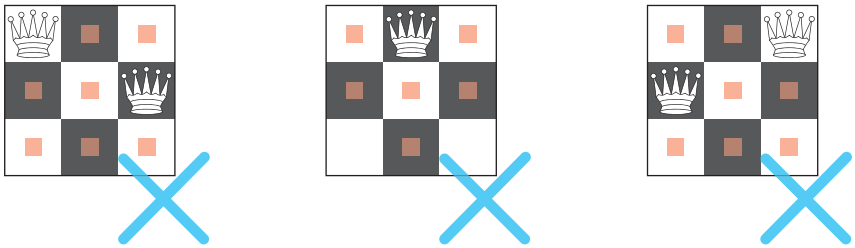
Es ist immer eine gute Idee, dass Problem zunächst einmal zu verkleinern.

Betrachten wir zuerst einmal ein Schachbrett von 3×3 Feldern und fragen nach den Möglichkeiten, drei Damen darauf zu platzieren, die sich gegenseitig nicht schlagen können. Wenn wir alle Möglichkeiten durchprobieren, wie drei (unterscheidbare) Damen auf die 9 Felder gestellt werden können, so sind dies $9 \cdot 8 \cdot 7 = 504$ Stellungen. Dabei entsprechen jeweils 6 Stellungen

der gleichen Situation, in welcher nur die drei Damen ihre Plätze untereinander vertauscht haben - bei ununterscheidbaren Damen haben wir noch $504 : 6 = 84$ Stellungen. Mit einer kleiner Überlegung können wir die Anzahl Elemente im Suchraum weiter einschränken: wir wissen, dass pro Zeile sicher nur eine Dame stehen darf, da zwei Damen sich ja gegenseitig schlagen würden. Nun hat es genau so viele Zeilen wie Damen, woraus folgt, dass pro Zeile genau eine Dame stehen muss. Wir werden daher also die erste Dame in die erste Zeile setzen, die zweite Dame in die zweite Zeile und die dritte dann in die unterste. Daher beschreiben wir eine Möglichkeit mit dem Tripel $(n_1, n_2, n_3),\ n_i \in \{1, 2, 3\}$. $(1, 2, 3)$ steht für die Möglichkeit, in welcher die oberste Dame im 1. Feld (ganz links), die 2. Dame im zweiten Feld (Mitte) und die unterste Dame im dritten Feld (rechts) gesetzt wird. Da keine zwei Damen in der gleichen Spalte stehen können, darf im Tripel (n_1, n_2, n_3) keine Zahl doppelt vorkommen: $n_i \neq n_j\ \forall i \neq j$, Damit haben wir noch $3 \cdot 2 \cdot 1 = 6$ Möglichkeiten, was eine erhebliche Erleichertung ist.



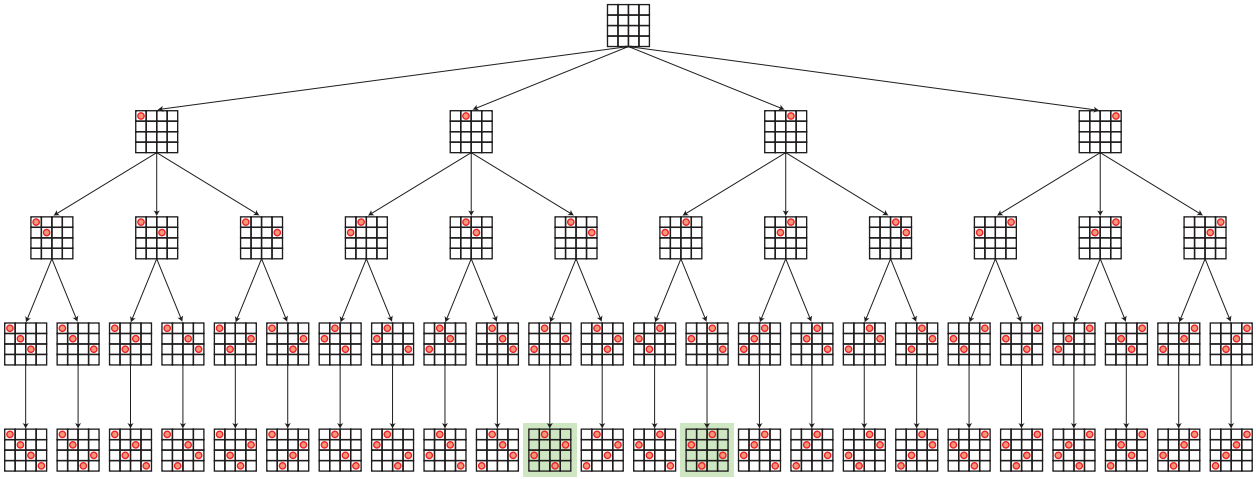
Beginnen wir links oben ($n_1 = 1$). Die Dame in der zweiten Reihe kann nun nur noch im Feld ganz rechts stehen, für die dritte Dame hat es aber keinen Platz mehr. Daraus schliessen wir, dass die oberste Dame nicht ganz links stehen kann ($n_1 \neq 1$). Also versuchen wir es einmal mit der mittleren Position ($n_1 = 2$). Die zweite Dame können wir nun aber schon nirgends mehr platzieren, also geht auch das nicht. Die Dame zu oberst ganz rechts ($n_1 = 3$) hinzustellen wird auch zu keiner Lösung führen, da diese Position symmetrisch zu der ganz links ist. Damit haben wir alle Möglichkeiten für n_1 durchlaufen und schliessen daraus, dass das Damenproblem für $n = 3$ (also 3 Damen auf einem 3×3 - Schachbrett) keine Lösung hat.



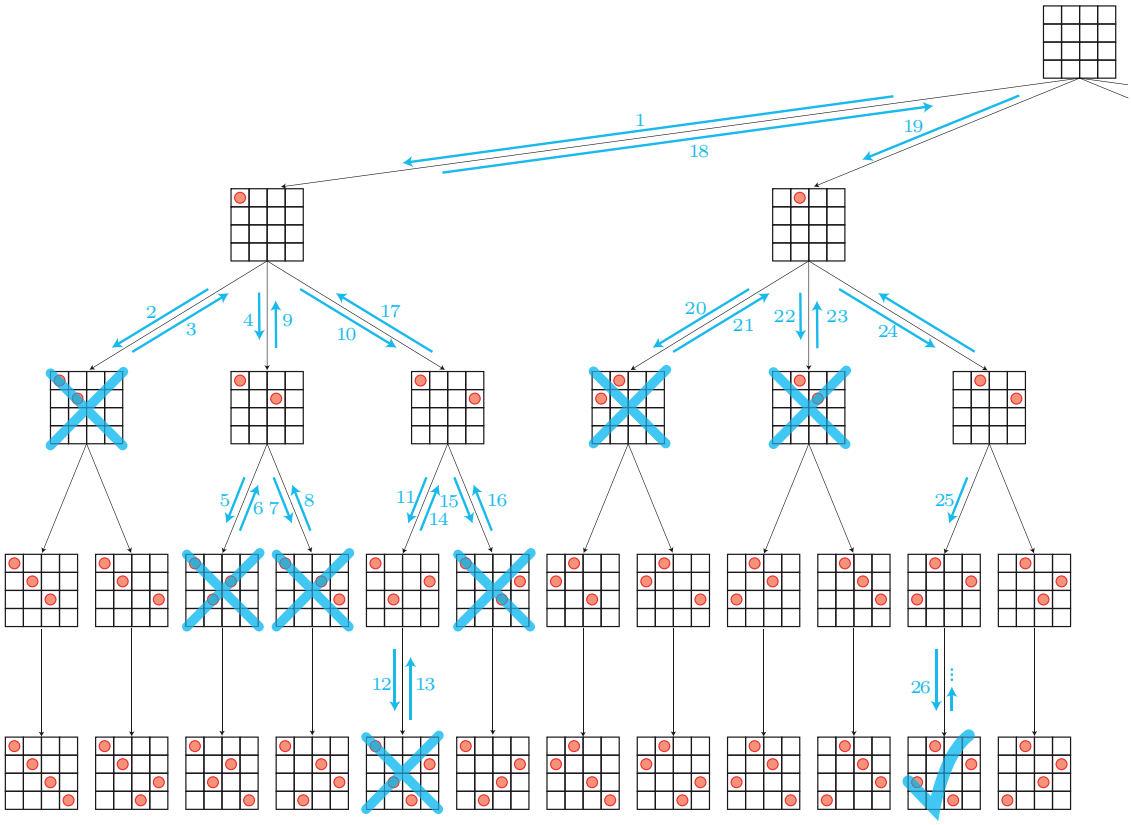
Aufgabe 4.1.1

Finden Sie von Hand alle Lösungen auf einem 4×4 - Brett.

Der Suchbaum zur Aufgabe ist auf der folgenden Seite dargestellt. Bitte blättern Sie erst um, wenn Sie die Aufgabe gelöst haben.

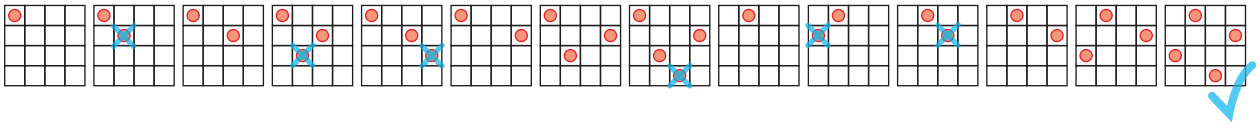


Es wäre nun natürlich möglich, für alle Blätter dieses Baums der Reihe nach durchzuprobieren, ob sie eine Lösung des Problems darstellen, das heisst im vorliegenden Fall, ob sich die Damen nicht gegenseitig bedrohen. Als wird uns den Fall $n = 3$ angesehen haben, sind wir allerdings etwas cleverer vorgegangen: sobald wird sicher sein konnten, dass die oberste Damen ganz links und die mittlere Damen in der Mitte nicht zulässig sind, haben wir erst gar nicht versucht, die unterste Damen zu setzen. Vielmehr haben wir die mittlere Dame wieder entfernt und ganz rechts hingestellt. Allgemein gehen wir so vor: Wir bauen die Möglichkeiten des Suchraums mit Hilfe des Baums auf. Erkennen wir in einem inneren Knoten, dass die entsprechende Teillösung nicht zulässig ist, so machen wir den letzten Schritt rückgängig, kehren also zum Vaterknoten zurück und nehmen dort die nächste Möglichkeit. Dieses Vorgehen ist durch den Namen "Backtracking" (oder auf Deutsch: Rückverfolgung) beschrieben, und ist für das Damenproblem mit $n = 4$ im folgenden Baum dargestellt (aus Platzgründen nur die linke Hälfte).



Backtracking ist das Durchlaufen eines Suchbaums, bei welchem Pfade nicht weiter verfolgt werden, von denen festgestellt werden kann, dass sie zu keiner Lösung führen werden. Statt den Pfad unnötig weiter zu verfolgen, wird ein Schritt zurück zum Vaterknoten gemacht, und von diesem aus weitergesucht.

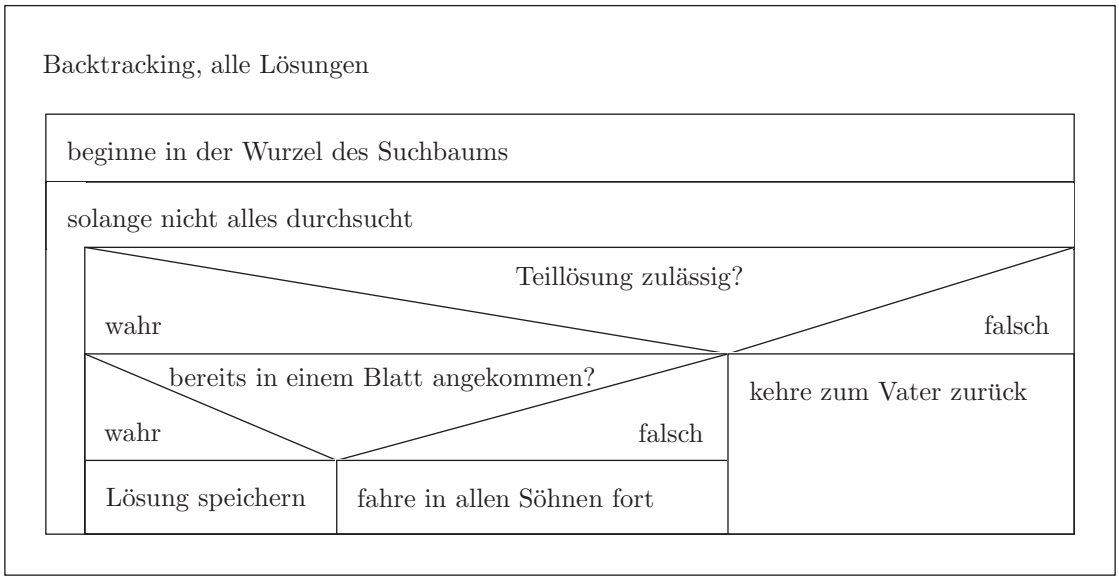
Sehen wir uns die Abfolge der betrachteten Konstellationen an, also derjenigen Knoten, bis zu denen die Äste des Baums verfolgt wurden:



Das Suchverfahren mittels Backtracking kann als doch einigermassen intelligent bezeichnet werden - es ähnelt auf jeden Fall dem beschriebenen Vorgehen für $n = 3$ Damen.

Abhängig davon, ob nur eine oder alle Lösungen gesucht sind, hören wir bei der ersten gefundenen Lösung auf oder aber registrieren diese nur und fahren im Baum weiter, indem wir einen Schritt zurück machen und im Baum weitersuchen.

Wir nehmen einmal an, dass alle Lösungen gesucht seien. Das Grundgerüst von Backtracking ist dann:



Aufgabe 4.1.2

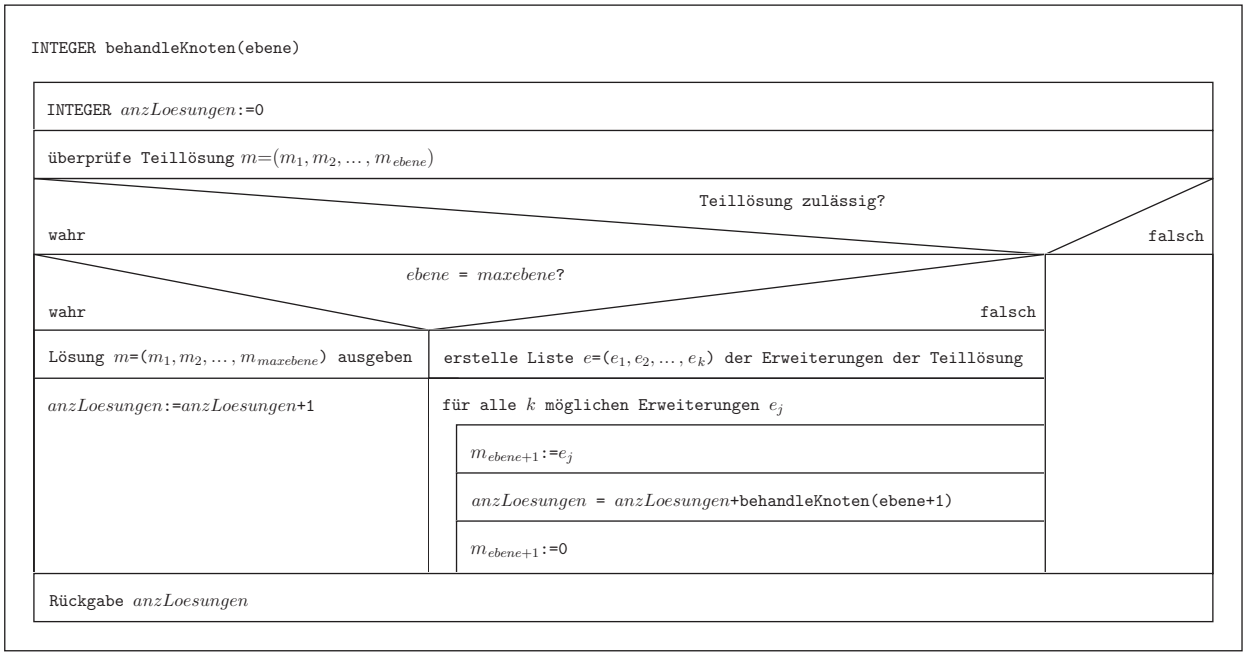
Wie sieht das Grundgerüst aus, wenn nur eine Lösung gesucht ist?

Aufgabe 4.1.3

Betrachten Sie nochmals den skizzierten Ablauf im linken Halbbaum des 4-Damen-Problems auf Seite 42. Geben Sie jeweils an, weshalb der Verlauf an der gefragten Stelle so ist wie gezeichnet.

a) $\xrightarrow{5}$ zu $\xrightarrow{6}$
b) $\xrightarrow{9}$ zu $\xrightarrow{10}$
c) $\xrightarrow{12}$ zu $\xrightarrow{13}$
d) $\xrightarrow{13}$ zu $\xrightarrow{14}$

In der obigen Kurzbeschreibung des Backtracking-Algorithmus kam die Anweisung "fahre der Reihe nach in allen Söhnen des Knotens weiter" vor und ebenso der Ausdruck "kehre zum Vater zurück". Am einfachsten lassen sich diese beiden Anweisungen durch Rekursion erreichen, also genau so, wie Sie es bereits im letzten Kapitel gemacht haben. Sollen der Reihe nach alle Söhne aufgerufen werden, so programmieren wir eine Schleife, erweitern darin den Lösungsvektor so, dass er jeweils einem Sohn entspricht, und rufen dann rekursiv die Abhandlungsfunktion eines Knotens auf, also die gleiche Funktion, in welcher wir uns gerade befinden. Um dann wieder zum Vater zurück zu kehren, machen wir die letzte Erweiterung des Lösungsvektors rückgängig und beenden die Funktion, so dass zur aufrufenden Funktion zurückgesprungen wird. Das entsprechende Struktogramm sieht also folgendermassen aus:



Aufgabe 4.1.4

Im Baumdiagramm auf Seite 42 entspricht jeder blaue Pfeil nach unten einem rekursiven Aufruf. Wie gross ist im 4-Damen-Problem die Einsparnis bei der Suche nach allen Lösungen, wenn Backtracking verwendet wird und nicht einfach jede Möglichkeit (also jedes Blatt des Baums) generiert und überprüft wird? (Der Aufruf der Ebene 0 soll dabei nicht gezählt werden.) Vergleichen Sie dazu die jeweils nötigen Schritte nach unten.

Vorgehen, um ein Problem mit Backtracking zu lösen

Um nun das 4-Damen-Problem mit Backtracking zu lösen, nehmen wir das im Struktogramm gezeigte Grundgerüst und überlegen uns noch Folgendes:

1. Wie kann eine Teillösung dargestellt werden?

   Wir setzen nacheinander die Damen und beginnen in der obersten Zeile. Wie schon beschrieben, speichern wir eine Teillösung in einem Vektor (Array) $\vec{n}$, dessen j-te Komponente n_j angibt, in welchem Feld sich die j-te Dame befindet. Dabei nummerieren wir die Felder von links her durch und beginnen mit der 1.

2. Wie wird der Baum effizient aufgebaut?

   Damit keine Dame in der gleichen Spalte wie eine schon gesetzte Dame steht, verlangen wir $n_j \neq n_i \quad \forall i < j$. Der Baum wird mit dieser Bedingung rekursiv aufgebaut.

3. Wie kann überprüft werden, ob eine Teillösung zulässig ist?

   Wir haben den Baum schon so aufgebaut, dass horizontal und vertikal sich keine Damen bedrohen. Es muss in jedem Schritt also noch überprüft werden, ob die neu hinzukommende Dame auf einer Diagonalen einer schon gesetzen Dame liegt. Die neu hinzukommende Dame wird in Spalte j auf das Feld n_j gesetzt. Damit darf eine Zeile darüber die Dame nicht auf Feld $n_j - 1$ oder $n_j + 1$ stehen und allgemein i Zeilen höher die Dame nicht auf Feld $n_j - i$ oder $n_j + i$ stehen.

Gestützt auf diese Überlegungen geben wir den Code für das 4-Damen-Problem an:

```
int[] m = new int[5];   // Lösung in m[1],...,m[4]


void setup()
{
  int anzL;

  anzL=behandleKnoten(0);
  println("Anzahl␣gefundener␣Lösungen␣"+anzL);
}

int behandleKnoten(int ebene)
{
  int anzL = 0;
  int i,j;
  boolean frei;

  if(pruefe(ebene)){
    if(ebene==4)
    {
```

```
      println("Lösung:␣"+join(nf(m, 0), ",␣"));
      anzL++;
    }
    else
    {
      for(i=1; i<=4; i++)
      {
        frei = true; // überprüfe, ob Spalte schon belegt
        for(j=1; j<=ebene; j++)
          frei = frei && (m[j]!=i);
        if(frei)
        {
          m[ebene+1]=i;
          anzL += behandleKnoten(ebene+1);
          m[ebene+1]=0; // unnötig, wird sowieso überschrieben
        }
      }
    }
  }
  return anzL;
}

boolean pruefe(int ebene)
{
  int i;
  boolean r=true;  // Teillösung zulässig?

  for(i=1; i<ebene; i++)
  {
    r = r && (m[ebene-i]!=m[ebene]-i);  // 1. Diagonale
    r = r && (m[ebene-i]!=m[ebene]+i);  // 2. Diagonale
  }
  return r;
}
```

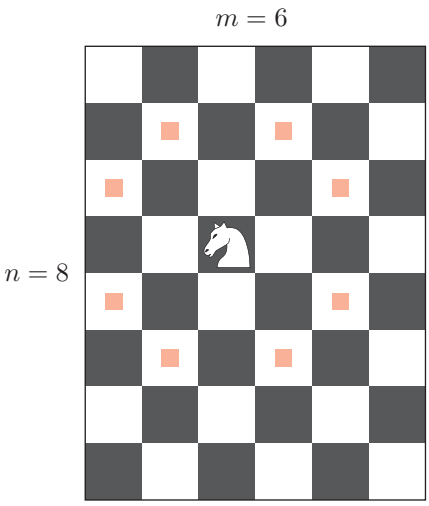
Aufgabe 4.1.5

Das obige Programm sucht die Lösungen des 4-Damen-Problems und gibt diese (als Lösungsvektor) aus. Zusätzlich zählt es auch die Anzahl der Lösungen.

a) Verallgemeinern Sie den Code, so dass er das n-Damen-Problem löst.
b) Wie viele Lösungen gibt es für das ursprüngliche Problem mit $n = 8$ Damen?
c) Wie viele sind es für $n = 10$, $n = 15$, ... ? Wo sehen Sie Probleme?

4.1.2 Das Springer-Problem

Ein anderes Schachproblem ist unter dem Namen "Springer-Problem" bekannt. Hier ist allerdings nur ein einziger Springer involviert, und seine Aufgabe ist es, einen Weg über das verallgemeinerte Schachbrett von m Feldern Breite und n Feldern Höhe zu finden, so dass er auf jedem Feld genau einmal vorbeikommt. Es gibt dabei verschiedene Varianten, so können nur geschlossene Wege gesucht werden, oder solche von einem festen Startpunkt aus, die dann irgendwo auf dem verallgemeinerten Schachbrett enden dürfen. Wir wollen hier für verschiedene Werte von m und n *eine* Lösung finden, die in der linken, oberen Ecke startet.



Um auf einen Algorithmus mit Backtracking zu kommen, sollten Sie sich wiederum folgende Fragen stellen:

1. Wie kann eine Teillösung dargestellt werden?
2. Wie wird der Baum effizient aufgebaut?
3. Wie kann überprüft werden, ob eine Teillösung zulässig ist?

Die Ausgabe der Lösung (sofern überhaupt eine existiert) kann als Text (beispielsweise als Folge der Koordinaten der angesprungenen Felder) oder grafisch geschehen. Die Wahl ist Ihnen überlassen.

Bevor Sie nun loslegen noch ein Tipp: Sie haben zwei Alternativen bei der Bestimmung der Felder, zu denen von einem momentanen Feld aus gesprungen werden kann.

1. Sie können zuerst alle möglichen Felder generieren, auf die vom momentanen Feld aus gesprungen werden kann. Dazu definieren Sie beispielsweise ein zweidimensionales Array `Moeglichkeiten[0..7][0..1]`, in welchem die x-Koordinate (in den Komponenten `moeglichkeiten[][0]`) und die y-Koordinate (in `moeglichkeit[][1]`) gespeichert wird - natürlich nur, wenn das entsprechende Feld nicht schon besucht worden ist. Neben dem Array brauchen Sie noch einen Zähler (beispielsweise `anzahlMoeglichkeiten`), der die Anzahl der wirklich vorliegenden Möglichkeiten aufnimmt. Das Array und der Zähler müssen lokal in der rekursiv aufgerufenen Funktion definiert sein, damit für jedes neu angesprungene Feld diese Informationen erzeugt und gespeichert werden, denn eventuell muss ein Schritt auf der betrachteten Stufe ja wieder rückgängig gemacht und die nächste der Möglichkeiten genommen werden.
2. Sie können auch immer alle 8 Möglichkeiten generieren und dann erst prüfen, ob das Feld wirklich auf dem Brett liegt und nicht schon belegt ist. In diesem Fall haben Sie immer eine Schlaufe über eben diese 8 Möglichkeiten.

Aufgabe 4.1.6

Schreiben Sie ein Programm, das für einen Springer auf einem $m \times n$-Brett einen möglichen Weg von der linken, oberen Ecke über alle Felder berechnet, so dass jedes Feld genau einmal

angesprungen wird. Die Ausgabe kann als Text oder als Grafik erfolgen. Berechnen Sie damit einen Weg des Springers auf einem 8×7-Brett und auch auf einem 10×10-Brett.

Aufgabe 4.1.7* Denksportaufgabe

Kann es einem 5×5-grossen Brett für den Springer einen **geschlossenen** Weg geben, der jedes Feld des Bretts genau einmal benutzt?

4.1.3 Symmetrische, magische Zahlenquadrate

Ein magisches Zahlenquadrat der Grösse $n \times n$ ist aus den Zahlen 1 bis n^2 so aufgebaut, dass jede Zahl genau einmal gebraucht wird und die Summe aller Zeilen und Spalten sowie der beiden Diagonalen alle gleich sind. Diese Summe wird als magische Zahl S_n bezeichnet und beträgt

16	9	5	4
3	6	10	15
2	7	11	14
13	12	8	1

$$S_n = \frac{1}{n}\left(\sum_{i=1}^{n^2} i\right) = \frac{1}{n}\frac{n^2 \cdot (n^2+1)}{2} = \frac{n^3+n}{2}$$

(da die Summe aller verwendeter Zahlen auf n Spalten verteilt werden muss.)

Verlangen wir gleichzeitig, dass jedes punktsymmetrisch zum Zentrum des Quadrats stehende Paar von Zahlen die gleiche Summe ergeben soll, so liegt ein symmetrisch, magisches Quadrat vor. Für symmetrische magische Quadrate beträgt diese Summe zweier punktsymmetrisch angeordneter Zahlen immer n^2+1, was für gerade n schnell nachprüfbar ist, denn die Summe S_n aller Zahlen wird auf $\frac{n^2}{2}$ Paare aufgeteilt. Ist n ungerade, so muss in der Mitte der Wert $\frac{n^2+1}{2}$ stehen.

Aufgabe 4.1.8

Erzeugen Sie ein Programm, das bei gegebenem n alle symmetrischen, magischen Zahlenquadrate bestimmt und ausgibt. Da neben einer Lösung immer auch noch weitere existieren, die durch Spiegelung(en) an der horizontalen oder vertikalen Achse oder auch an der 1. Diagonalen entstehen, soll nur ein Vertreter gezählt und dargestellt werden. Wir halten uns an die Frénicle-Standardform, die verlangt, dass

1. die Zahl in der linken, oberen Ecke die grösste Eckzahl ist,
2. die Zahl rechts neben der linken, oberen Ecke kleiner ist als die Zahl direkt unter dieser Ecke.

Bestimmen Sie die Anzahl symmetrischer, magischer Quadrate für $n = 3, 4, 5$.

Schon für relativ kleine n gibt es so viele Lösungen, dass ihre Auszählung sehr lange dauert. Auch wenn Sie Ihr Programm so ändern, dass es Ihnen nur **eine** Lösung sucht, kann die Antwort sehr lange auf sich warten lassen. Für dieses Problem ist Backtracking zwar anwendbar, da es aber schnellere (konstruktive) Methoden gibt, magische Quadrate zu generieren, wird man das Backtracking höchstens dafür verwenden, zu festem n die totale Anzahl Lösungen zu bestimmen.

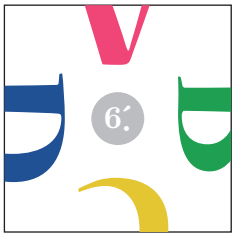
4.1.4 Das verflixte Puzzlespiel

Aufgabe 4.1.9

Mit dem Puzzle-Spiel haben Sie sich in Kapitel 3.1 bereits auseinandergesetzt und sich dort überlegt, wie der Suchbaum generiert werden kann. Bauen Sie darauf auf und schreiben Sie ein Programm, das Ihnen berechnet, auf wie viele Arten und wie die Karten richtig zusammengesetzt werden können. Im Anhang (S. 67) finden Sie die Karten, die Sie ausschneiden und mit denen Sie die berechnete Lösung dann überprüfen können.

Aufgabe 4.1.10

Ersetzen Sie die Karte 6 durch 6' und berechnen Sie dafür nun wieder alle Möglichkeiten, die Karten richtig aneinanderlegen zu können. Sie finden die Karte 6' ebenfalls im Anhang.



4.2 Backtracking mit Heuristiken

Dieses Unterkapitel muss nicht bearbeitet werden. Es handelt von einer Möglichkeit, wie die Suche nach einer Lösung beschleunigt werden kann.

4.2.1 Nochmals das Springerproblem, für grosse m, n

Sollen in einem Suchproblem alle Lösungen gefunden werden, so bleibt nichts anderes übrig, als den entsprechenden Suchbaum möglichst effizient (klein) aufzubauen und dann vollständig abzuarbeiten. Wichtig ist dabei, dass möglichst früh erkannt werden kann, wenn eine Teillösung zu keiner Lösung mehr führen kann. Interessiert jedoch nur eine Lösung, so kann man versuchen, durch heuristische Suchkriterien eine solche schneller zu finden. Heuristisch heisst dabei, dass mit beschränktem Wissen in möglichst kurzer Zeit ein Erfolg verzeichnet werden soll. Heuristiken müssen also weder wissenschaftlich bewiesen noch in jedem Fall vorteilhaft sein. Vielmehr sind sie mehr oder weniger intuitiv getroffene Annahmen, von denen erhofft wird, dass sie schneller zum Erfolg, also zu einer Lösung führen.

Nehmen wir nochmals das Springer-Problem aus Kap. 4.1.2. Auch wenn wir nur eine einzige Lösung haben wollen, so kann dies für $m = n = 10$ schon ziemlich lange dauern, bis diese gefunden ist. Wählen wir aus den Möglichkeiten, von momentanen Feld aus weiterzuspringen, allerdings immer die "beste", so sind wir schnell fertig. Die Frage ist natürlich, woran wir die "beste" Möglichkeit erkennen können.

Aufgabe 4.2.1

Überlegen Sie sich einige Strategien, in welcher Reihenfolge die Sprungmöglichkeiten durchprobiert werden sollen, um schneller zu einer Lösung zu kommen. In anderen Worten: Geben Sie mögliche Heuristiken an.

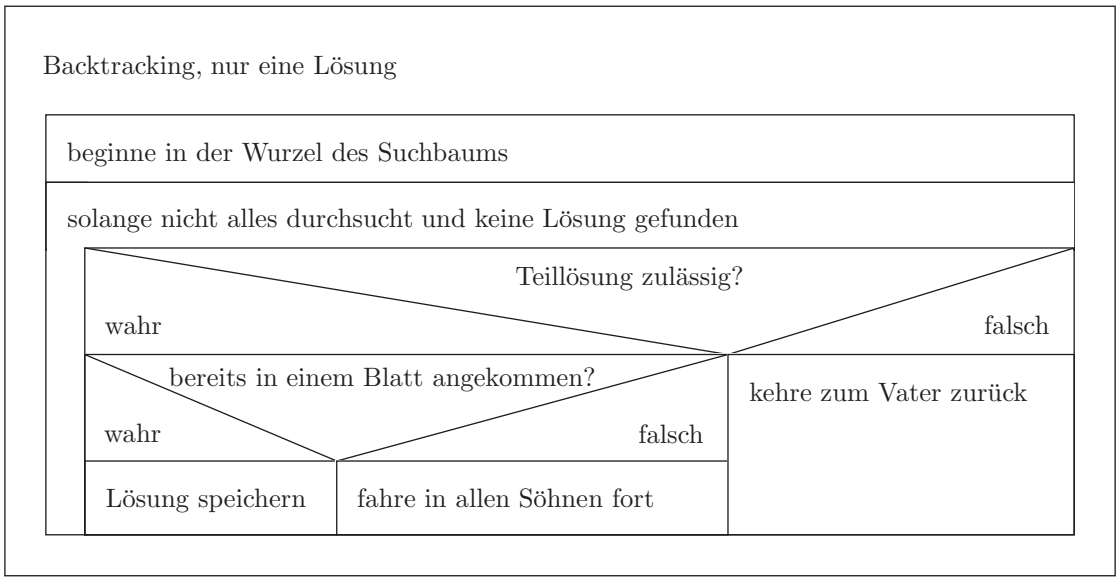
4.3 Lösungen der Aufgaben des Kapitels

Lösung zu Aufgabe 4.1.1

Die Lösung zu Aufgabe 4.1.1 ist im Text erklärt.

Lösung zu Aufgabe 4.1.2

Der einzige Unterschied zum bereits behandelten Algorithmus ist das Abbruchkriterium der Schlaufe:



Lösung zu Aufgabe 4.1.3

a) Die Zwischenlösung $(1, 3, 2)$ ist nicht zulässig, daher wird einen Schritt zurück gegangen.
b) Die Überprüfung des zweiten Sohns von (1) ist abgeschlossen und nun folgt diejenige des dritten (und letzten) Sohns.
c) Die Möglichkeit $(1, 4, 2, 3)$ ist nicht zulässig, daher wird einen Schritt zurück gegangen.
d) Alle Möglichkeiten in $(1, 4, 2)$ wurden durchprobiert, daher wird zum Knoten $(1, 4)$ zurückgekehrt.

Lösung zu Aufgabe 4.1.4

Mit Backtracking gibt es im ganzen Baum 32 nach unten zeigende Pfeile (15 bis zur Lösung, in linken Teilbaum dann noch einen einzigen weiteren, im rechten Teilbaum sind es (aus Sym-

metriegründen) nochmals so viele). Ohne Backtracking (also bei einer vollen "Tiefensuche") wären es 64. Der Aufwand hat sich also halbiert, die Einsparung beträgt 50 %.

(Für $n = 8$ sind es mit Backtracking 5508 Aufrufe, gegenüber von $8 + 8 \cdot 7 + 8 \cdot 7 \cdot 6 + \ldots + 8! = 8(1 + 7(1 + 6(1 + \ldots 2(1 + 1)) \ldots))) = 109\,600$ Aufrufen ohne Backtracking, was einer Einsparung von 95 % entspricht.)

Lösung zu Aufgabe 4.1.5

a)

```
final int N = 8;
int[] m = new int[N+1];  // Lösung in m[1],...,m[N]

void setup()
{
  int anzL;
  String s;

  anzL=behandleKnoten(0);
  println("Anzahl gefundener Lösungen "+anzL);
}

int behandleKnoten(int ebene)
{
  int anzL = 0;
  int i,j;
  boolean frei;

  if(pruefe(ebene)){
    if(ebene==N){
      println("Lösung: "+join(nf(m, 0), ", "));
      anzL++;
    }
    else{
      for(i=1; i<=N; i++){
        frei = true; // überprüfe, ob Spalte schon belegt
        for(j=1; j<=ebene; j++)
          frei = frei && (m[j]!=i);
        if(frei){
          m[ebene+1]=i;
          anzL += behandleKnoten(ebene+1);
          m[ebene+1]=0; // unnötig, wird sowieso überschrieben
        }
      }
    }
  }
  return anzL;
}
```

```
boolean pruefe(int ebene)
{
  int i;
  boolean r=true;  // Teillösung zulässig?

  for(i=1; i<ebene; i++){
    r = r && (m[ebene-i]!=m[ebene]-i);  // 1. Diagonale
    r = r && (m[ebene-i]!=m[ebene]+i);  // 2. Diagonale
  }
  return r;
}
```

b) Für $n = 8$ sind es 92 Lösungen.

c) $n = 10$ ergibt 724 Lösungen, $n = 15$ führt auf 2 279 184 Lösungen, wobei diese besser nicht mehr alle ausgegeben werden. Der Baum wächst mindestens exponentiell; für grosse n dauert die Berechnung daher sehr lange.

Zur Skalierung:

Die Zahl der rekursiven Aufrufe wächst mit n stark an, wie die folgende Tabelle zeigt. Zwar ist es nicht ganz so schlimm, wie wenn wirklich jede der $n!$ Möglichkeiten blind durchprobiert würden, dennoch ist das Wachstum mindestens exponentiell.

n	Anzahl rekursiver Aufrufe	Anzahl Lösungen
2	4	0
3	11	0
4	32	2
5	101	10
6	356	4
7	1 345	40
8	5 508	92
9	24 011	352
10	110 004	724
11	546 357	2 680
12	2 915 740	14 200
13	16 469 569	73 712
14	99 280 504	365 596
15	636 264 861	2 279 184

Lösung zu Aufgabe 4.1.6

Mögliche Antworten auf die drei Fragen zur Umsetzung des Backtrackings für das Springer-Problem:

1. Wie kann eine Teillösung dargestellt werden?

   Die Folge der angesprungenen Felder wird in einem Array `zug[M*N][2]` gespeichert, wobei `zug[i][0]` die x-Koordinate und `zug[i][1]` die y-Koordinate des Felds vor dem i-ten Zug aufnimmt.

2. Wie wird der Baum effizient aufgebaut?

   Auf jedem Feld wird über alle maximal acht Möglichkeiten verzweigt, wie weitergesprungen werden kann.

3. Wie kann überprüft werden, ob eine Teillösung zulässig ist?

   Existiert von einem Feld aus keine Möglichkeit mehr um weiterzuspringen, so ist die Teillösung unzulässig, und der letzte Schritt muss rückgangig gemacht werden.

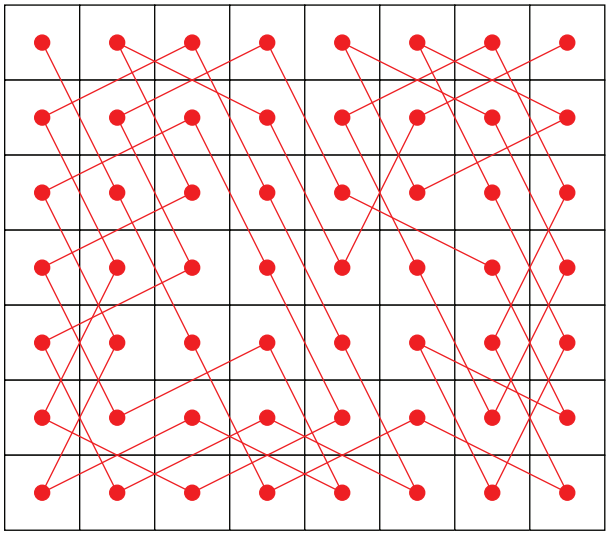
Eine mögliche (sowohl grafische als auch textliche) Lösung ist:

```
static int N=7;
static int M=8;
int rand = 20;
int fg = 50;
int[][] zug = new int [N*M][2];
boolean[][] feld = new boolean[M][N];  // feld[x][y]
int moves[][] =
   {{1,2},{2,1},{2,-1},{1,-2},{-1,-2},{-2,-1},{-2,1},{-1,2}};
boolean loesungGefunden=false;


void setup()
{
  int i,j;

  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      feld[i][j] = false;
  zug[0][0] = 0; zug[0][1] = 0; // beginne oben links
  feld[0][0] = true;
  loesungGefunden = springe(1);
  if(loesungGefunden)
    println("Lösung␣gefunden");
  else
    println("keine␣Lösung␣vorhanden");
  size(M*fg+2*rand, N*fg+2*rand);
}

void draw()
{
  int i, j;

  if(loesungGefunden)
  {
    stroke(0);
    fill(255);
    for(i=0; i<M; i++)
      for(j=0; j<N; j++)
```

```
      {
        rect(i*fg+rand, j*fg+rand, fg, fg);
      }
    stroke(255,0,0);
    fill(255,0,0);
    for(i=0; i<M*N-1; i++)
    {
      ellipse(rand+zug[i][0]*fg+0.5*fg, rand+zug[i][1]*fg+0.5*fg,
         0.2*fg, 0.2*fg);
      line(rand+zug[i][0]*fg+0.5*fg, rand+zug[i][1]*fg+0.5*fg, rand+
         zug[i+1][0]*fg+0.5*fg,  rand+zug[i+1][1]*fg+0.5*fg);
    }
     ellipse(rand+zug[M*N-1][0]*fg+0.5*fg, rand+zug[M*N-1][1]*fg
        +0.5*fg, 0.2*fg, 0.2*fg);
  }
}

boolean springe(int ebene)
{
  int anzMoeglichkeiten=0;
  int[][] moeglichkeiten = new int[8][2];
  int i,x,y;

  if(ebene==N*M) // Lösung
  {
    print("Lösung:␣");
    for(i=0; i<N*M-1; i++)
      print("("+zug[i][0]+","+zug[i][1]+"),");
    print('\n');
    return true;
  }
  // else
  for(i=0; i<8; i++)  // versuche alle 8 Richtungen
  {
    x = zug[ebene-1][0]+moves[i][0];
    y = zug[ebene-1][1]+moves[i][1];
    if(abs(x%M) == x && abs(y%N)==y && feld[x][y]==false)
    {
      moeglichkeiten[anzMoeglichkeiten][0] = x;
      moeglichkeiten[anzMoeglichkeiten][1] = y;
      anzMoeglichkeiten++;
    }
  }
  for(i=0; i<anzMoeglichkeiten; i++)
  {
    x=moeglichkeiten[i][0];
    y=moeglichkeiten[i][1];
    zug[ebene][0]=x;
    zug[ebene][1]=y;
    feld[x][y]=true;
    if(springe(ebene+1))
      return true;
```

```
    feld[x][y]=false;
  }
  return false;
}
```

Für $m = 8$ und $n = 7$ berechnet das obige Programm die folgende Lösung:



Die Lösung für ein Brett mit 10×10 Feldern lässt lange auf sich warten. Eine mögliche Beschleunigung besprechen wir im Kapitel 4.2.

Lösung zu Aufgabe 4.1.7

In einem geschlossenen Weg ist der Anfangspunkt willkürlich wählbar, wir können wieder die linke, obere Ecke nehmen, die wir als "weiss" annehmen. Zu dieser Ecke müssen wir am Schluss, also nach 25 ($= 5 \cdot 5$) Zügen, wieder zurückkehren. Bei jedem Zug aber wechselt die Farbe des Feldes. Nach einem Zug werden wir also auf einem "schwarzen" Feld sein, nach zwei Zügen dann wieder auf einem "weissen". Nach 25 Zügen müssten wir also wieder auf ein schwarzes Feld kommen, gleichzeitig aber auch auf das weisse Startfeld. Dies ist unmöglich.

Lösung zu Aufgabe 4.1.8

```
static int N=5;
static boolean AUSGABE=false;
int[][] quadrat = new int [N][N];
boolean[] zahl = new boolean[N*N+1];  // zahl[i]=true: i schon
   gebraucht.
int anzL=0;
int Sp = N*N+1;
int S = N*Sp/2;
int[] zeile = new int[N];
int[] spalte = new int[N];
int[] diagonale = new int[2];
```

```
void setup()
{
  int i,j;

  for(i=1; i<=N*N; i++)
    zahl[i] = false;
  if(N%2==1) // ungerade
  {
    quadrat[N/2][N/2]=Sp/2;
    berechneSummen(N/2, N/2, 1);
    zahl[Sp/2]=true;
  }
  magQuadrat(0);
  println("es␣gibt␣"+anzL+"␣symmetrische,␣magische␣("+N+","+N+")␣
     Quadrate");
}



void magQuadrat(int ebene)
{
  int x,y, i, j;

  x = ebene % N;
  y = ebene / N;

  if(ebene==N*N/2)
  {
    anzL++;
    if(AUSGABE)
    {
      println("Lösung␣"+anzL);
      for(j=0; j<N; j++)
      {
        for(i=0; i<N; i++)
          print(quadrat[i][j]+"␣");
        print('\n');
      }
    }
    return;
  }
  // else
  for(i=N*N; i>0; i--)  // gehe über alle Zahlen (wegen Standardform
      absteigend)
  {
    if(zahl[i]==false)
    {
      quadrat[x][y] = i;
      berechneSummen(x,y,1); // Summen anpassen
      quadrat[N-x-1][N-y-1] = Sp-i;
      berechneSummen(N-x-1,N-y-1,1);
      if(check(x,y) && check(N-x-1, N-y-1))
```

```
      {
        zahl[i]=true;
        zahl[Sp-i]=true;
        magQuadrat(ebene+1);
        zahl[i]=false;
        zahl[Sp-i]=false;
      }
      berechneSummen(x,y,-1);  // Zahl aus Summe wieder entfernen
      quadrat[x][y] = 0;
      berechneSummen(N-x-1,N-y-1,-1);
      quadrat[N-x-1][N-y-1] = 0;
    }
  }
}

void berechneSummen(int x, int y, int faktor)
{
  zeile[y] += faktor*quadrat[x][y];
  spalte[x] += faktor*quadrat[x][y];
  if(x==y)
    diagonale[0] += faktor*quadrat[x][y];
  if(x==N-y-1)
    diagonale[1] += faktor*quadrat[x][y];
}

boolean check(int x, int y)  // überprüft, ob Element in (x,y)
   zulässig ist (true)
{
  if(zeile[y] > S || spalte[x] > S)  // Summe der Zeile oder der
     Spalte zu hoch
    return false;
  if(diagonale[0] > S)  // 1. Diagonale
    return false;
  if(diagonale[1]> S)  // 2. Diagonale
    return false;
  if(x==N-1 && y ==0 && quadrat[x][y]>quadrat[0][0])  // prüfe auf
     Standardform
    return false;
  if(x==0 && y == N-1 && quadrat[x][y]>quadrat[0][0])
    return false;
  if(x==N-1 && y == N-1 && quadrat[x][y]>quadrat[0][0])
    return false;
  if(x==0 && y==1 && quadrat[x][y] > quadrat[y][x])
    return false;
  return true;
}
```

Resultate:

Es gibt 1 symmetrisches, magisches 3×3 Quadrat, 48 symmetrische, magische 4×4 Quadrate und 48 544 symmetrische, magische 5×5 Quadrate.

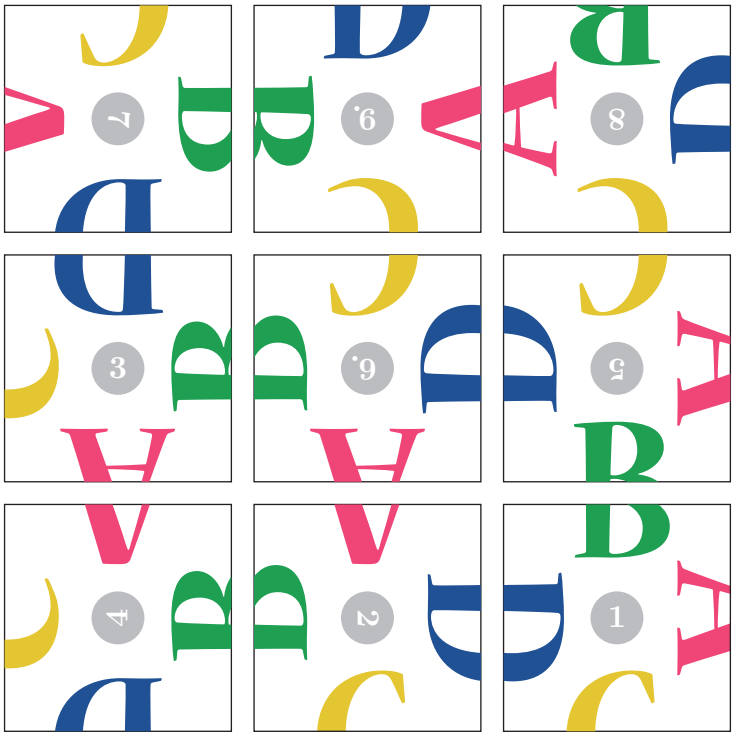
Lösung zu Aufgabe 4.1.9

```
static int Au=1, Bu=2, Cu=3, Du=4, Do=5, Co=6, Bo=7, Ao=8;

int[][] karte = {{Au,Bu,Do,Co},{Co,Du,Ao,Bu},{Bo,Do,Cu,Au},
                  {Ao,Cu,Du,Bo}, {Du,Bo,Au,Co}, {Bu,Au,Do,Co},
                  {Du,Bu,Co,Ao}, {Au,Cu,Do,Bo}, {Bo,Cu,Ao,Du}};
int[][][] belegung = new int[3][3][2];   // [i][j][0]  Kartennummer.
    [i][j][1]: Drehung (*90°)

void setup()
{
  setzeKarte(0);
}

void setzeKarte(int platz)
{
  int i,j, jmax, m, n;
  boolean frei, passt;
  int iplatz = platz % 3;
  int jplatz = platz / 3;

  for(i=0; i<9; i++)   // gehe über alle 9 Karten
  {
    frei = true;
    for(j=0; j<platz; j++)
      frei =  frei && (belegung[j%3][j/3][0] != i);
    if(frei)  // noch nicht gelegt
    {
      jmax = (i==0 ? 0 : 3); // Karte 0 nur in eine Richtung, um
         nicht gedrehte Lösungen zu erhalten
      for(j=0; j<=jmax; j++)  // gehe über alle vier Drehungen
      {
        passt = true;
        if(iplatz !=0) // muss mit horizontalem Vorgänger passen
          passt = passt && (karte[i][(6-j)%4] + karte[belegung[
             iplatz-1][jplatz][0]][(4-belegung[iplatz-1][jplatz][1])
             %4] == 9);
        if(jplatz != 0)  // muss mit vertikalem Vorgänger passen
          passt = passt && (karte[i][(5-j)%4] + karte[belegung[
             iplatz][jplatz-1][0]][(3-belegung[iplatz][jplatz-1][1])
             %4] == 9);
        if (passt)   // Karte passt
        {
          belegung[iplatz][jplatz][0]=i;  // Kartennummer
          belegung[iplatz][jplatz][1]=j;  // Drehung
          if(platz<8)  // noch nicht alle Karten gelegt
            setzeKarte(platz+1);
          else
          {
            println("Lösung␣gefunden!");
            for(n=0; n<3; n++)
            {
```

```
                for(m=0; m<3; m++)
                  print("Karte␣"+(belegung[m][n][0]+1) +",␣gedreht␣um␣
                     "+belegung[m][n][1]*90+"°,␣");
                print('\n');
              }
            }
          }
        }
      }
    }
  }
```

Es gibt genau eine Lösung:

Karte 7, gedreht um 270°, Karte 9, gedreht um 180°, Karte 8, gedreht um 180°,
Karte 3, gedreht um 0°, Karte 6, gedreht um 180°, Karte 5, gedreht um 180°,
Karte 4, gedreht um 90°, Karte 2, gedreht um 270°, Karte 1, gedreht um 0°.



Lösung zu Aufgabe 4.1.10

Es gibt drei Lösungen:

Karte 2, gedreht um 180°, Karte 3, gedreht um 270°, Karte 4, gedreht um 0°,
Karte 8, gedreht um 270°, Karte 7, gedreht um 0°, Karte 1, gedreht um 0°,
Karte 9, gedreht um 270°, Karte 5, gedreht um 270°, Karte 6', gedreht um 180°.

Karte 8, gedreht um 90°, Karte 2, gedreht um 270°, Karte 1, gedreht um 0°,
Karte 9, gedreht um 180°, Karte 3, gedreht um 270°, Karte 4, gedreht um 0°,
Karte 5, gedreht um 180°, Karte 6', gedreht um 90°, Karte 7, gedreht um 0°.

Karte 9, gedreht um 180°, Karte 3, gedreht um 270°, Karte 4, gedreht um 0°, Karte 5, gedreht um 180°, Karte 6', gedreht um 90°, Karte 7, gedreht um 0°, Karte 1, gedreht um 0°, Karte 2, gedreht um 0°, Karte 8, gedreht um 90°.

Lösung zu Aufgabe 4.2.1

Mögliche Heuristiken:

- Es sollen primär diejenigen Felder angesprungen werden, die Nahe am Rand liegen.
- Felder, die wenig Möglichkeiten zum Weiterspringen haben, sollen zuerst angesprungen werden.
- ...

Wir werden die folgende Heuristik anwenden: für jede der maximal 8 Sprungmöglichkeiten berechnen wir die Anzahl der noch möglichen Züge, von diesen weiterzukommen, und speichern dies in einem Array mit Namen `bewertung[]`. Wir nehmen zuerst diejenige mit der kleinsten Bewertung, danach die mit der nächst grösseren und so weiter. Dies erreichen wir (ziemlich primitiv), indem aus allen Bewertungen die kleinste suchen, diese Möglichkeit nehmen, und danach diese Bewertung so hoch setzen, dass in der nächsten Suche nach dem kleinsten Element alle noch nicht genommenen Möglichkeiten sicher eine kleinere Bewertung haben werden. Den schon bestehenden Code zur Suche einer Lösung ohne Heuristik ergänzen wir also durch eine Funktion zur Bewertung, welche die Anzahl der Möglichkeiten zählt, weiterzukommen, und die Schleife, in welcher die Möglichkeiten rekursiv aufgerufen werden, um eine Sortierung, welche diejenigen mit einer kleinen Bewertung bevorzugt. Damit erhalten wir den folgenden Code

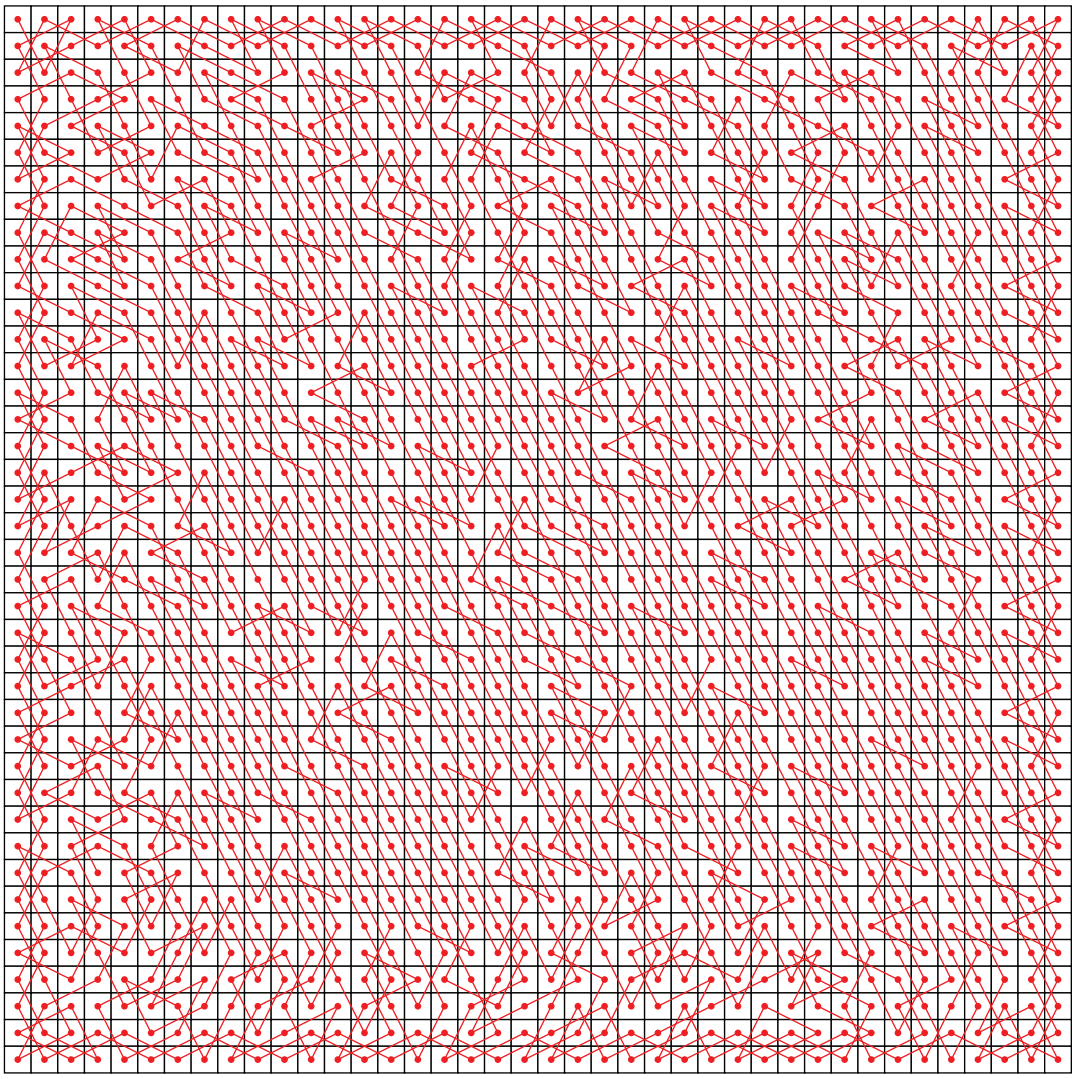
```
static int N=40;
static int M=40;
int rand = 20;
int fg = 50;
int[][] zug = new int [N*M][2];
boolean[][] feld = new boolean[M][N];  // feld[x][y]
int moves[][] =
   {{1,2},{2,1},{2,-1},{1,-2},{-1,-2},{-2,-1},{-2,1},{-1,2}};
boolean loesungGefunden=false;

void setup()
{
  int i,j;

  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      feld[i][j] = false;
  zug[0][0] = 0; zug[0][1] = 0; // beginne oben links
  feld[0][0] = true;
  loesungGefunden = springe(1);
  if(loesungGefunden)
    println("Lösung␣gefunden");
```

```
  else
    println("keine Lösung vorhanden");
  size(M*fg+2*rand, N*fg+2*rand);
}

void draw()
{
  int i, j;

  if(loesungGefunden)
  {
    stroke(0);
    fill(255);
    for(i=0; i<M; i++)
      for(j=0; j<N; j++)
      {
        rect(i*fg+rand, j*fg+rand, fg, fg);
      }
    stroke(255,0,0);
    fill(255,0,0);
    for(i=0; i<M*N-1; i++)
    {
      ellipse(rand+zug[i][0]*fg+0.5*fg, rand+zug[i][1]*fg+0.5*fg,
         0.2*fg, 0.2*fg);
      line(rand+zug[i][0]*fg+0.5*fg, rand+zug[i][1]*fg+0.5*fg, rand+
         zug[i+1][0]*fg+0.5*fg, rand+zug[i+1][1]*fg+0.5*fg);
    }
    ellipse(rand+zug[M*N-1][0]*fg+0.5*fg, rand+zug[M*N-1][1]*fg+0.5*
       fg, 0.2*fg, 0.2*fg);
  }
}

boolean springe(int ebene)
{
  int anzMoeglichkeiten=0;
  int[][] moeglichkeiten = new int[8][2];
  int i,j, x,y;
  int minBew, mini;
  int[] bewertung = new int[8];

  if(ebene==N*M) // Lösung
  {
    print("Lösung: ");
    for(i=0; i<N*M-1; i++)
      print("("+zug[i][0]+","+zug[i][1]+"),");
    print('\n');
    return true;
  }
  // else
  for(i=0; i<8; i++)  // versuche alle 8 Richtungen
  {
    x = zug[ebene-1][0]+moves[i][0];
```

```
    y = zug[ebene-1][1]+moves[i][1];
    if(abs(x%M) == x && abs(y%N)==y && feld[x][y]==false)
    {
      moeglichkeiten[anzMoeglichkeiten][0] = x;
      moeglichkeiten[anzMoeglichkeiten][1] = y;
      bewertung[anzMoeglichkeiten] = weitereMoeglichkeiten(x,y);
      anzMoeglichkeiten++;
    }
  }
  for(i=0; i<anzMoeglichkeiten; i++)
  {
    minBew = bewertung[0];
    mini = 0;
    for(j=1; j<anzMoeglichkeiten; j++)
      if(bewertung[j]<minBew)
      {
        minBew = bewertung[j];
        mini = j;
      }
    x=moeglichkeiten[mini][0];
    y=moeglichkeiten[mini][1];
    zug[ebene][0]=x;
    zug[ebene][1]=y;
    feld[x][y]=true;
    if(springe(ebene+1))
      return true;
    feld[x][y]=false;
    bewertung[mini]=M*N;
  }
  return false;
}

int weitereMoeglichkeiten(int x,int y)
{
  int i, zaehler=0;
  int xn, yn;

  for(i=0; i<8; i++)  // versuche alle 8 Richtungen
  {
    xn = x+moves[i][0];
    yn = y+moves[i][1];
    if(abs(xn%M) == xn && abs(yn%N)==yn && feld[xn][yn]==false)
      zaehler++;
  }
  return zaehler;
}
```

Es ist verblüffend, wie diese doch recht simple Heuristik auch ein relativ grosses Brett von 40×40 Feldern schnell löst. Tatsächlich ist die Limitierung nun nicht mehr die Zeit zur Ausführung des Programms sondern die Stack-Grösse, die bei einem 50×50-Brett mit 2500 ineinandergeschachtelten, rekursiven Aufrufen zu einer Fehlermeldung führt. Für grössere

Felder müsste iterativ vorgegangen werden, vergleichen Sie dazu Kapitel 3.4.

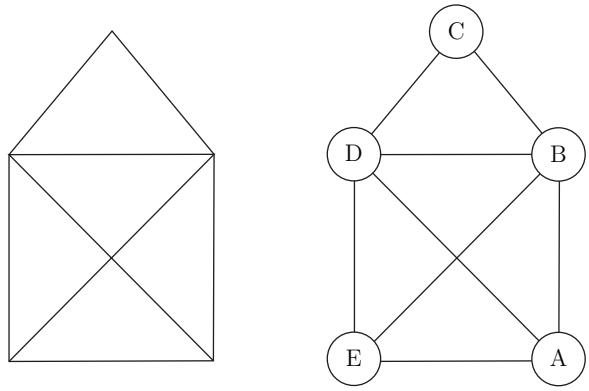


4.4 Kapiteltest III

Dieser Kapiteltest besteht aus zwei grösseren Projekten: dem Haus des Nikolaus und einem Sudoku-Löser. Wegen des grösseren Umfangs sind die Aufgaben in Unterkapiteln untergebracht.

4.4.1 Das Haus des Nikolaus

Ein altes Kinderspiel ist es, zum Spruch "das ist das Haus des Ni - ko - laus" mit einem Strich, also ohne abzusetzen, die nebenstehende Figur zu zeichnen.



Wir können dieses Haus als (ungerichteten) Graphen auffassen, wozu wir die Ecken mit A, B, C, D und E bezeichnen und nun nach einem Weg fragen, der jede Kante genau einmal enthält. Sehen wir uns die Grade der Ecken an, so sehen wir, dass sowohl E als auch A den Grad 3 (also ungeraden Grad) haben und die anderen Ecken einen geraden Grad. Daher muss es einen solchen Weg geben, und die Anfangs- und Endpunkte sind E und A.

Natürlich gehen wir hier einen Schritt weiter als nur eine Möglichkeit zu finden, wie das Haus gezeichnet werden kann. Vielmehr interessiert hier, wie viele Möglichkeiten es total sind.

Aufgabe 4.4.1

Schreiben Sie ein Programm, dass die Anzahl Wege von E nach A berechnet, welche jede Kante genau einmal beinhalten.

Es kann hilfreich sein, die Information des Graphen in der Adjazenzmatrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

zu speichern. Das Element $(A)_{ij}$ gibt an, wie viele Kanten von Ecke i zu Ecke j existieren.

Wird im Backtracking dann eine Kante gebraucht, so kann sie aus der Adjazenzmatrix entfernt werden. Wird sie (in einem Rückschritt) wieder freigegeben, hält man dies ebenfalls in der Adjazenzmatrix wieder fest.

4.4.2 Sudoku

Es ist nun an der Zeit, wieder zur ersten Fragestellung, dem Sudoku, zurückzukommen. Sie haben nun alles erlernt, so dass Sie auch diese Aufgabe mit Hilfe eines Programms lösen können. Wie Sie bei den magischen Quadraten schon gesehen haben, kann die zur Lösung benötigte Zeit ziemlich lange sein. Wenn Sie den Suchbaum in einem Sudoku aber geschickt aufbauen, ist die Lösung innert Sekundenbruchteilen berechnet! Die Kunst ist also, den Suchbaum so zu generieren, dass er möglichst wenig Äste hat.

Aufgabe 4.4.2

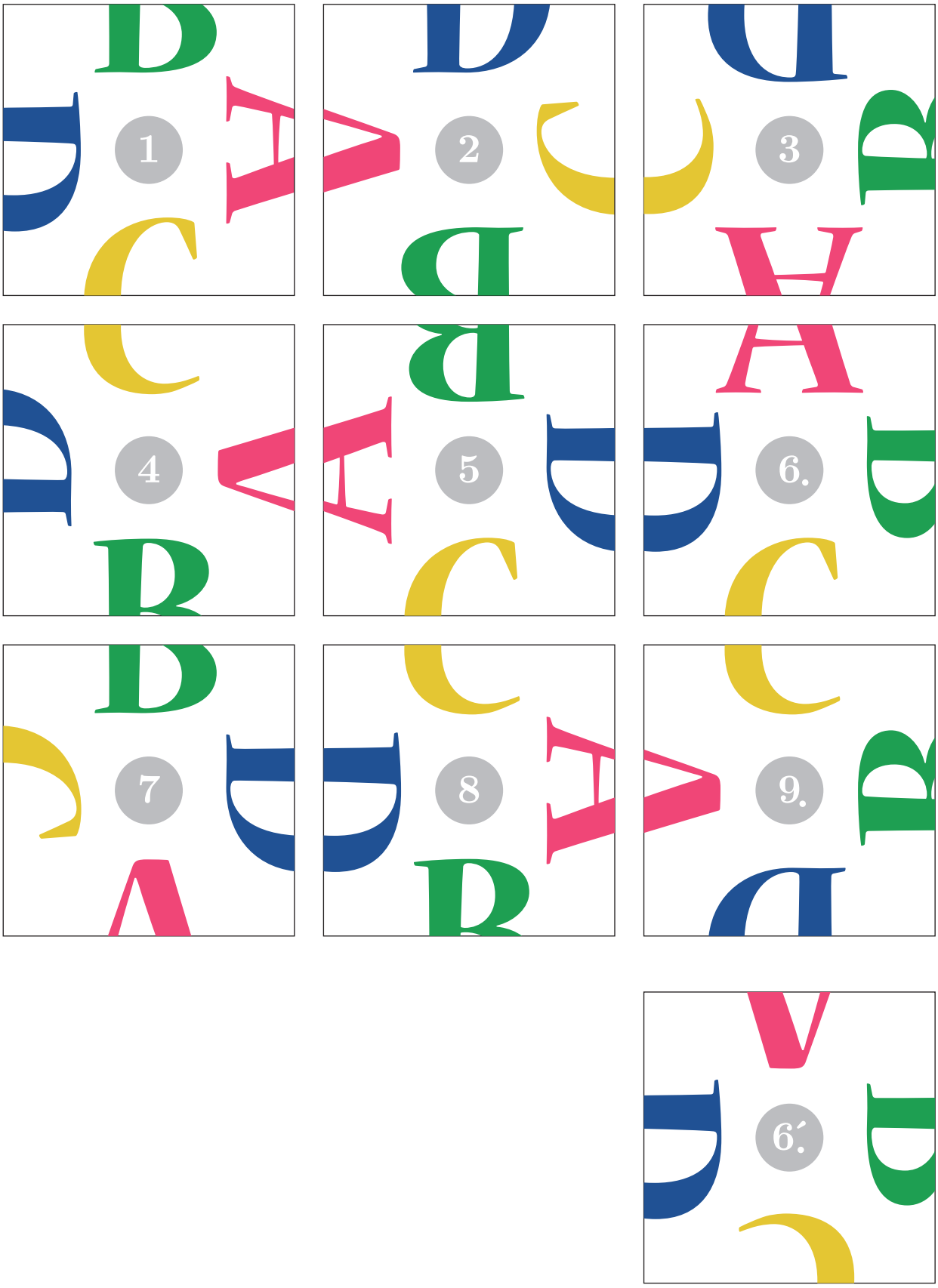
Schreiben Sie ein Programm, dass das Sudoku von Seite 5 löst. Damit Sie nicht immer umblättern müssen, ist es hier nochmals abgebildet:

			1	7			2	
2					9	7		
	7		2	8	3	4		
	5	7						
3		9				8		7
						5	3	
		8	3	1	6		4	
		5	7					8
	4			5	8			

*

Anhang A

A.1 Karten zum "verflixten Puzzle"



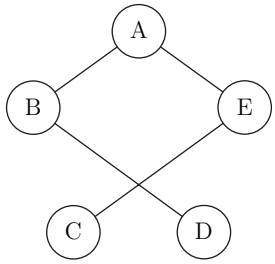
A.2 Grundbegriffe zu (ungerichteten) Graphen

Einfacher Graph

Ein einfacher Graph besteht aus einer Menge V von Knoten, $V = \{v_i\}$, und einer Menge E von Kanten, $E = \{e_j\}$, wobei eine Kante jeweils zwei verschiedene Knoten verbindet: $e_i = (v_m, v_n)$ mit $m \neq n$.

Beispiel:

Einfacher Graph $G(V, E)$ mit $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ und $E = \{(\mathrm{A}, \mathrm{B}), (\mathrm{A}, \mathrm{E}), (\mathrm{B}, \mathrm{D}), (\mathrm{C}, \mathrm{E})\}$.

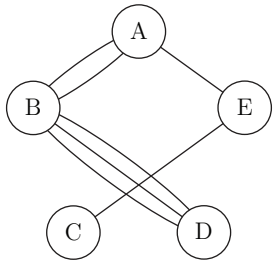


Mehrfachkanten

Eine Mehrfachkante kann entstehen, wenn zwischen zwei Knoten auch mehr als nur eine Kante zugelassen wird.

Beispiel:

Graph $G(V, E)$ mit Mehrfachkanten zwischen A und B (doppelte Kante) sowie zwischen B und D (dreifache Kante).

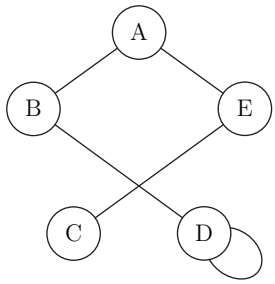


Schlinge

Führt eine Kante wieder zum gleichen Knoten zurück, so handelt es sich um eine Schlinge.

Beispiel:

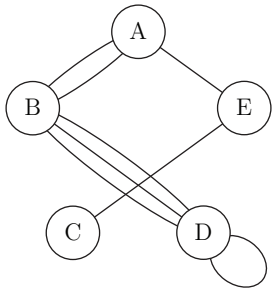
Graph $G(V, E)$ mit Schlinge im Knoten D.



Grad eines Knotens

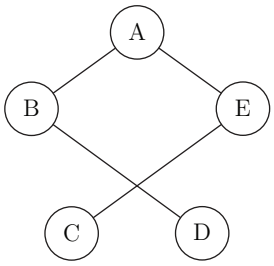
Mit dem Grad wird angegeben, wie viele Kantenansätze im betrachteten Knoten vorliegen. Eine Schlinge erhöht den Grad um zwei, da beide Ansätze gezählt werden.

Grad(A) = 3, Grad(B) = 5, Grad(C) = 1, Grad(D) = 5, Grad(E) = 2

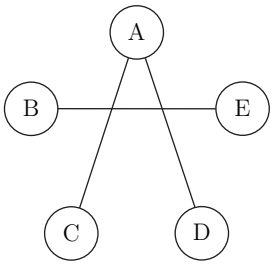


Zusammenhängender Graph

Kann von jedem beliebigen Punkt des Graphen aus jeder andere über eine oder mehrere Kanten erreicht werden, so ist er Graph zusammenhängend.



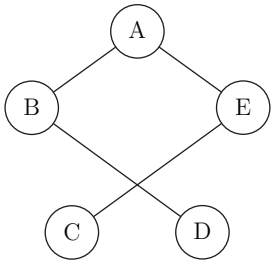
zusammenhängender Graph

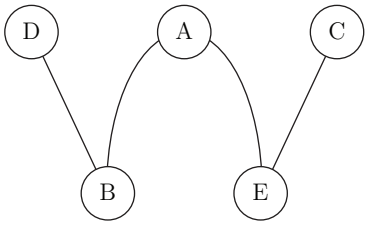


nicht zusammenhängender Graph

Freiheiten in der grafischen Darstellung

Ein Graph ist durch die Angabe seiner Knoten und Kanten vollständig definiert. Wird er gezeichnet, so bleibt die Wahl der Anordnung der Knoten und der Form der Kanten. Zwei gleiche Graphen müssen daher nicht unbedingt schon auf den ersten Blick als solche erkennbar sein, wie das folgende Beispiel zeigt:



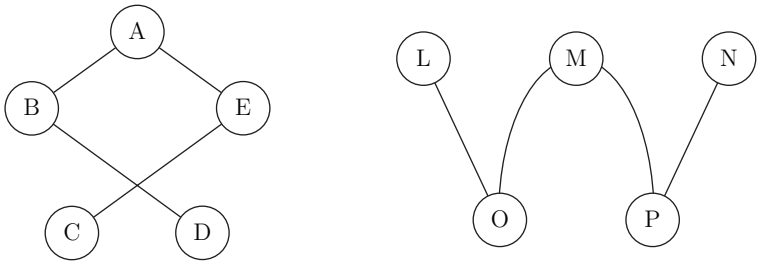


Isomorphe Graphen

Unterscheiden sich zwei Graphen höchstens in der Benennung ihrer Knoten, so sind sie isomorph.

Beispiel:

Die beiden folgenden Graphen sind isomorph. Die eineindeutige Beziehung der Knoten ist A↔M, B↔O, C↔N, D↔L, E↔P.



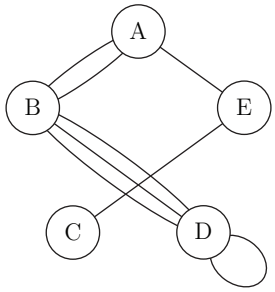
Adjazenzmatrix

Ein Graph kann durch Angabe seiner Adjazenzmatrix $A = (a_{ij})$ vollständig beschrieben werden. Dazu werden die Zeilen i und die Spalten j mit den Knotennamen beschriftet. Das Element $a_{ij} = a_{ij}$ gibt jeweils an, wie viele Kanten die Knoten i und j verbinden.

Beispiel:

Die Adjazenzmatrix des nebenstehenden Graphen ist

	A	B	C	D	E
A	0	2	0	0	1
B	2	0	0	3	0
C	0	0	0	0	1
D	0	3	0	1	0
E	1	0	1	0	0



A.3 Das Rosinenbrötchen-Problem (Wahrscheinlichkeitsrechnung)

Die folgende Aufgabe zeigt die Grenzen der einfachen Betrachtung durch einen Suchbaum auf. Mit einigen Überlegungen kann sie aber dennoch durch ein Programm gelöst werden.

Ein Bäcker macht aus einem Teig, in dem sich 60 Rosinen befinden, 20 Brötchen. Dabei nehmen wir an, dass jede Rosine mit der gleichen Wahrscheinlichkeit zu jedem der Brötchen kommen kann. Wie gross ist nun die Wahrscheinlichkeit, dass mindestens ein Brötchen keine Rosine hat?

Schreiben Sie ein Programm, dass die gesuchte Wahrscheinlichkeit berechnet.

Lösung:

1. Lösungsidee über vollen Suchbaum

Da für jede Rosine entschieden werden kann, zu welchem der 20 Brötchen sie gehören soll, liegt ein 60-stufiger Baum vor, in welchem jeder Knoten den Ausgangsgrad 20 hat. Alle so erzeugten Ergebnisse sind gleichwahrscheinlich, also kann die Anzahl der Ergebnisse, in denen ein Brötchen keine Rosine bekommen hat, durch die gesamte Anzahl der Möglichkeiten geteilt werden. Allerdings ist letztere grösser als 10^{78} und damit praktisch unberechenbar. Eine so erzeugte Möglichkeit hätte aber die Form $(r_1, r_2, \ldots, r_{60})$ mit $r_i \in \{1, 2, \ldots 20\}$; r_i gibt also an, welches Brötchen die i-te Rosine erhalten hat.

2. Lösungsidee

Wir müssen Äste des Lösungsbaums zusammenfassen. Dazu verwenden wir ein ähnliches Programm wie schon für die Quersummen-Aufgabe, jetzt aber für 20-stellige Zahlen im 61er-System (so dass jede Stelle die Werte 0 bis 60 aufnehmen kann). Die so erzeugten Möglichkeiten sind aber nicht gleichwahrscheinlich, da die Reihenfolge nicht beachtet wurde. Die Möglichkeit $(b_1, b_2, \ldots, b_{20})$ gibt an, wie viele Rosinen in Brot 1, in Brot 2, ... sind. Eine Möglichkeit $(b_1, b_2, \ldots, b_{20})$ entspricht also allen Möglichkeiten $(r_1, r_2, \ldots, r_{60})$, in denen b_1 mal ein $r_j = 1$ war, b_2 mal ein $r_j = 2$, und so fort. Dies entspricht der Anzahl aller 60-stelliger Zeichenfolgen mit b_1 Einsen, b_2 Zweien, ... Also steht die Möglichkeit $(b_1, b_2, \ldots, b_{20})$ für genau

$$\binom{60!}{b_1! \cdot b_2! \cdot b_3! \cdot b_{20}!} \text{ gleichwahrscheinlichen Ergebnisse } (r_1, r_2, \ldots, r_{60})$$

Wir generieren also alle Möglichkeiten $(b_1, b_2, \ldots, b_{20})$, multiplizieren sie mit $\binom{60!}{b_1! \cdot b_2! \cdot b_3! \cdot b_{20}!}$ und zählen sowohl die totale Anzahl der Möglichkeiten A_{tot} wie auch die Anzahl A_{betr}, in welchen mindestens ein $r_j = 0$ ist. Die gesuchte Wahrscheinlichkeit ergibt sich dann als der Quotient der beiden Werte

$$P = \frac{A_{betr}}{A_{tot}}$$

Da der Faktor 60! in jedem Summanden und dann auch im Quotient vorkommt, lassen wir ihn in der Berechnung gleich weg.)

Leider benötigt auch dieses Programm zu lange.

3. Lösungsidee

Aufbauend auf der vorhergehenden Idee fassen wir nochmals Äste zusammen. So bestimmen wir nur die ungeordneten Möglichkeiten $(b_1, b_2, \ldots, b_{20})$ mit $b_i \leq b_{i-1}$ $(2 \leq i \leq 20)$. Diese multiplizieren wir dann mit allen Möglichkeiten, die b_i verschieden anzuordnen. Gibt es $n_1, n_2, \ldots, n_k$ Mal die gleiche Werte b_i, so wird mit

$$\binom{20!}{n_1!n_2!\ldots n_k!}$$

multipliziert.

Das so funktionierende Programm ist

```
static int N = 20;
static int ANZROSINEN = 60;
double zaehlerBetr = 0;
double zaehlerTot = 0;
double m;

int[] zahl = new int[N]; // global
double[] rezfak = new double[ANZROSINEN+1];  // nehmen die Werte 1/k
   ! auf
double[] fak = new double[N+1];  // nehmen die Werte k! auf

void setup()
{
  initialisiereRezFak();
  initialisiereFak();
  generiereZiffernfolge(0,0);
  println("Wahrscheinlichkeit␣=␣"+((float) (zaehlerBetr))/zaehlerTot
     );
}

void initialisiereRezFak()
{
  int i;
  rezfak[0]= 1;
  for(i=1; i<=ANZROSINEN; i++)
    rezfak[i] = rezfak[i-1]/i;
}

void initialisiereFak()
{
  int i;
  fak[0]= 1;
  for(i=1; i<=N; i++)
    fak[i] = fak[i-1]*i;
}

void generiereZiffernfolge(int momTiefe, int momRosinensumme)
{
  int i, imax, j, rosDiff;
```

```
  boolean kommtNichtVor;

  if(momTiefe == N-1)  // hinterste Stelle, die geforderte Quersumme
      muss gebildet werden
  {
    rosDiff = ANZROSINEN - momRosinensumme;
    if(rosDiff > zahl[momTiefe-1]) // nicht erlaubt
      return;
    zahl[momTiefe] = rosDiff;
    // berechne den Faktor m, mit dem diese Möglichkeit
       multipliziert werden muss
    // m = QUERSUMME!/(zahl[0]! zahl[1]! ... zahl[N-1]!), lassen
       QUERSUMME! weg
    m = 1.;
    for(j=0; j<N; j++)
      m *= rezfak[zahl[j]];
    // multipliziere mit Anzahl geordneter Müglichkeiten für zahl[]
    m *= fak[N];
    i=0;
    while(i<N)
    {
      imax = zahl[i];
      j=0;
      while(i<N && zahl[i]==imax)
      {
        j++;
        i++;
      }
      m *= rezfak[j];
    }
    zaehlerTot += m;
    kommtNichtVor = false;  // hat ein Brötchen keine Rosinen?
    for(j=0; j<N; j++)
      kommtNichtVor = (kommtNichtVor || (zahl[j]==0));
    if(kommtNichtVor)
      zaehlerBetr += m;
  }
  else
  {
    rosDiff = ANZROSINEN - momRosinensumme;
    if(momTiefe>0)
      imax = min(rosDiff, zahl[momTiefe-1]); // höchstens so gross
         wie Vorgänger
    else
      imax = ANZROSINEN;
    for(i=0; i<=imax; i++)
    {
      zahl[momTiefe] = i;
      generiereZiffernfolge(momTiefe+1, momRosinensumme+i);
    }
  }
}
```

Die damit berechnete Wahrscheinlichkeit dafür, dass bei 60 Rosinen zufällig auf 20 Brötchen verteilt mindestens eines keine Rosinen hat, ist 0.639395.

Nicht ganz so genau, aber um einiges schneller erhalten wir das Resultat auch mit einer grösseren Menge automatisch ausgeführter Pseudozufallsexperimente, in welchem die Rosinen (pseudo-)zufällig auf die Brote verteilt werden und dann überprüft wird, ob mindestens ein Brötchen keine Rosine hat. Ein solches Programm ist beispielsweise

```
int N = 20;
int ANZROSINEN = 60;
int M = 100000;  // Anzahl Pseudozufallsexperimente
int i, j;
int zaehlerBetr = 0;
int[] zahl = new int[N];
boolean kommtNichtVor;

for(i=0; i<M; i++)
{
  for(j=0; j<N; j++)
    zahl[j]=0;
  for(j=0; j<ANZROSINEN; j++)
    zahl[(int) (random(N))]++;
  kommtNichtVor = false;
  for(j=0; j<N; j++)
    kommtNichtVor = (kommtNichtVor || (zahl[j]==0));
  if (kommtNichtVor)
    zaehlerBetr++;
}
println("Wahrscheinlichkeit: "+((float) zaehlerBetr)/M);
```

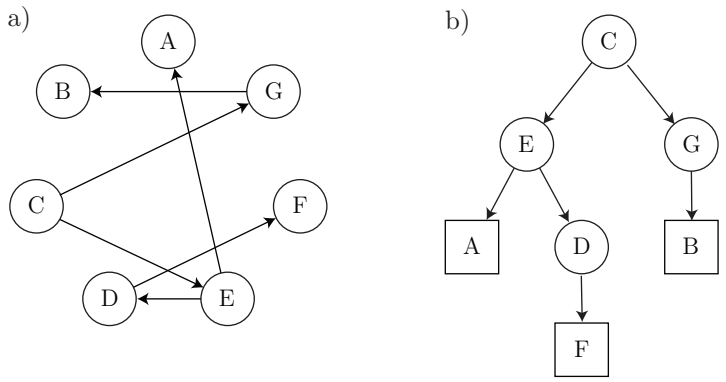
und ergibt eine Wahrscheinlichkeiten von ≈ 0.64.

A.4 Lösungen zum Kapiteltest I

Lösung zu Aufgabe 2.8.1

a) Der linke Graph ist gerichtet, der rechte ungerichtet.
b) Eingangsgrad(E) = 2, Ausgangsgrad(E) = 1
c) beide Graphen sind zyklisch.
Kreise im linken Graphen sind (B,E,B) und (B,C,D,E,B).
Der einzige Kreis im rechten Graphen ist (A,E,B,D,A).

Lösung zu Aufgabe 2.8.2

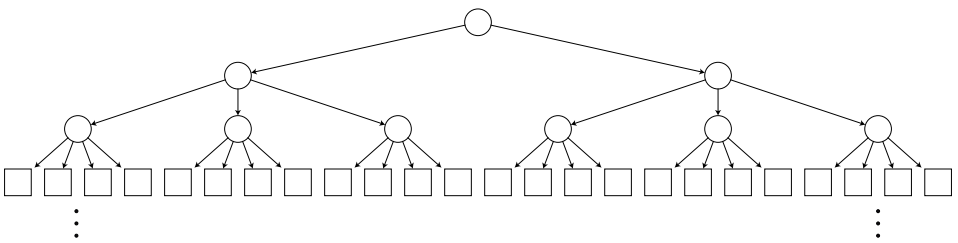


Lösung zu Aufgabe 2.8.3

Ein innerer Knoten in Tiefe n hat n Geschwister und daher $n + 2$ Söhne. Der Baum mit der höchsten Anzahl Knoten entsteht, falls alle Blätter in der maximalen Tiefe sind. Er hat dann 1 Wurzel, 2 Knoten in Tiefe 1, $2 \cdot 3 = 3!$ Knoten in Tiefe 2, $2 \cdot 3 \cdot 4 = 4!$ Knoten in Tiefe 3, $5!$ Knoten in Tiefe 4 und $6!$ Blätter in Tiefe 5. Total sind es also

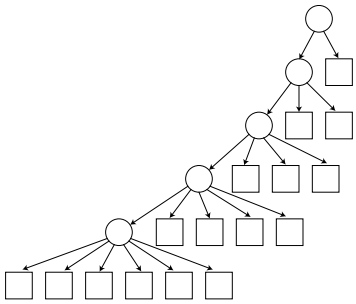
$$\sum_{i=1}^{6} i! = 873 \text{ Knoten}$$

(Im folgenden Bild ist der Baum bloss bis zur dritten Stufe dargestellt, er enthält $\sum_{i=1}^{4} i! = 33$ Knoten.)



Der Baum mit der kleinsten Anzahl Knoten entsteht, wenn pro Stufe nur ein einziger Knoten Söhne hat (die anderen Knoten schon Blätter sind). Es sind dann

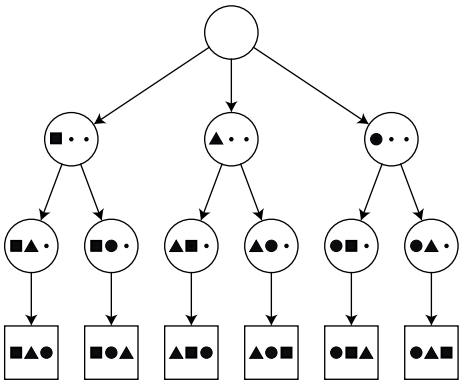
$$1 + 2 + 3 + 4 + 5 + 6 = 21 \text{ Knoten}$$



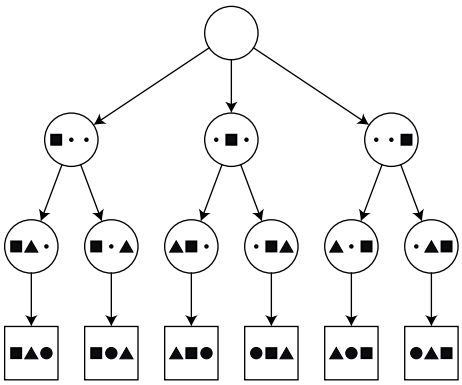
A.5 Lösungen zum Kapiteltest II

Lösung zu Aufgabe 3.7.1

Es gibt zwei Herangehensweisen. Entweder wird zuerst die Stelle ganz links, dann die in der Mitte und zuletzt die ganz rechts belegt:



oder aber es wird zuerst das Quadrat auf eine der drei offenen Stellen verteilt, dann das Dreieck auf eine der beiden noch in Frage kommenden und zum Schluss der Kreis auf die letzte Stelle



Lösung zu Aufgabe 3.7.2

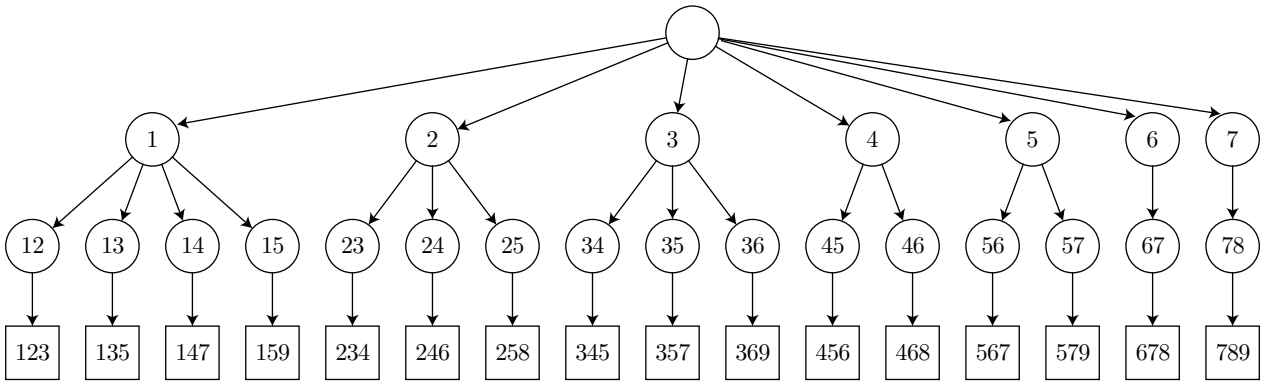
a)

```
static int N = 5;
int zaehler = 0;
int[] zahl = new int[N];

void setup()
{
  int i;
  for(i=0; i<N; i++)
    zahl[i] = 0;
  generiereZahlenfolge(1);
  println("Es␣gibt␣"+zaehler+"␣Zahlen␣mit␣der␣gefordeten␣
     Eingeschaft");
}

void generiereZahlenfolge(int momZahl) // setze momZahl Zahl an
    eine der noch offenen Stellen
{
  int i;

  for(i=0; i<N; i++)
    if(zahl[i]==0)  // Schlaufe über offene Stellen
    {
      zahl[i]=momZahl;
      if(momZahl < N)
        generiereZahlenfolge(momZahl+1);
      else
      {
        println(join(nf(zahl,0),",␣"));  // Ausgabe des Arrays
        zaehler++;
      }
      zahl[i]=0;  // unbedingt wieder frei geben
    }
}
```

b) Es gibt $n!$ Möglichkeiten.

Lösung zu Aufgabe 3.7.3

a)



b)

```
static int N = 3;
int zaehler = 0;

void setup()
{
  generiereZiffernfolge(0,1);
  println("Es␣gibt␣"+zaehler+"␣Zahlen␣mit␣der␣gefordeten␣
     Eingeschaft");
}
```

(der Code geht auf der nächsten Seite weiter)

```
void generiereZiffernfolge(int tempZahl, int momTiefe)
{
  int i, j, step;

  if(momTiefe==1)  // erste Ziffer
  {
    for(i=1; i<=9-(N-1); i++)
      generiereZiffernfolge(i, momTiefe+1);
  }
  else if(momTiefe==2)  // zweite Ziffer
  {
    j = tempZahl % 10;  // letzte bisherige Ziffer
    for(i=j+1; i<=9-(N-2)*(i-j); i++)
      generiereZiffernfolge(10*tempZahl+i, momTiefe+1);
  }
  else  // Schrittweite zwischen Ziffern schon fest
  {
    step = (tempZahl% 10) - (tempZahl % 100)/10;  // Schrittweite
    i = (tempZahl%10)+step;
    if(momTiefe == N)
    {
      zaehler++;
       println("Lösung␣"+zaehler+"␣:␣"+(tempZahl*10+i));
    }
    else
      generiereZiffernfolge(10*tempZahl+i, momTiefe+1);
  }
}
```

Lösung zu Aufgabe 3.7.4

```
static int anzStellen = 5;
int i;
int[] ziffer = new int[anzStellen];
int summeZiffern = 0;
int anzM = 0;

void setup()
{
  for(i=0; i<anzStellen; i++)
  {
    ziffer[i]=1;
    summeZiffern += ziffer[i];
  }
  while(ziffer[0]<10)
  {
    if(summeZiffern % 3 == 0) // Quersumme durch 3 teilbar
    {
      for(i=0; i<anzStellen; i++)   // Ausgabe
        print(ziffer[i]);
      println("␣");
      anzM++;
    }
    ziffer[anzStellen-1] += 2;  // hinterste Ziffer um 2 erhöhen
    summeZiffern += 2;  // Quersumme anpassen
    i=anzStellen-1;
    while(ziffer[i]>9 && i>0)   // Überlauf der Stellen korrigieren
    {
      ziffer[i]=1;  // von 11 auf 1 korrigieren
      ziffer[i-1] += 2;  // vordere Stelle um 2 erhöhen
      summeZiffern -= 8;  // Quersumme vermindert sich um 8
      i--;
    }
  }
  println("total:␣"+anzM+"␣Möglichkeiten");
}
```

Es gibt 1042 Möglichkeiten.

A.6 Lösungen zum Kapiteltest III

Lösung zu Aufgabe 4.4.1

```
final int N = 8;  // 8 Kanten
final int L = 5;  // 5 Ecken
int[][] adj
   ={{0,1,0,1,1},{1,0,1,1,1},{0,1,0,1,0},{1,1,1,0,1},{1,1,0,1,0}};
   // Adjazenzmatrix
int[] weg = new int[N+1];
char[] knoten = {'E','D','C','B','A'};
int anzLoesungen = 0;

void setup()
{
  weg[0] = 0; // beginne in A
  noLoop();
}

void draw()
{
  HdN(0);
  println("Anzahl␣Lösungen:␣"+anzLoesungen);
}

boolean HdN(int ebene)
{
  int anzMoegl;
  int[]  moegl = new int[L];
  boolean akz = false;
  int i,j;

  j = 0;
  for(i=0; i<L; i++)
    if(adj[weg[ebene]][i] > 0)
    {
      moegl[j] = i;
      j++;
    }
  anzMoegl=j;
  if(ebene == N)  // Haus fertig
  {
    anzLoesungen++;
    print("Lösung␣"+anzLoesungen+"␣:␣");
    for(i=0; i<N+1; i++)
      print(knoten[weg[i]]);
    print('\n');
    akz = true;
  }
  else
  {
   for(i=0; i<anzMoegl; i++)
```

```
    {
        weg[ebene+1]=moegl[i];
        adj[weg[ebene]][weg[ebene+1]] -= 1;  // beide Einträge der Adj
           -Matrix löschen
        adj[weg[ebene+1]][weg[ebene]] -= 1;
        if(HdN(ebene+1))
          akz = true;
        adj[weg[ebene]][weg[ebene+1]] += 1;  // beide Einträge der Adj
           -Matrix löschen
        adj[weg[ebene+1]][weg[ebene]] += 1;
     }
   }
   return akz;
}
```

Lösung zu Aufgabe 4.4.2

Der Algorithmus ist schnell, wenn jeweils das Feld mit den wenigsten Einsetzmöglichkeiten gesucht wird.

```
final int N=3; // maximale Anz. Lösungen
int[][] zahlen =
{{0,0,0,  1,7,0,  0,2,0},
 {2,0,0,  0,0,9,  7,0,0},
 {0,7,0,  2,8,3,  4,0,0},
//-------------------
 {0,5,7,  0,0,0,  0,0,0},
 {3,0,9,  0,0,0,  8,0,7},
 {0,0,0,  0,0,0,  5,3,0},
//-------------------
 {0,0,8,  3,1,6,  0,4,0},
 {0,0,5,  7,0,0,  0,0,8},
 {0,4,0,  0,5,8,  0,0,0}};
int vorgegebeneZahlen;
int anzLoesungen=0;

void setup()
{
  vorgegebeneZahlen = lese();
  if(!konsistent())
    println("nicht␣konsistent");
  else if(vorgegebeneZahlen < 81)
       loese(vorgegebeneZahlen);
  else
    println("schon␣alle␣Zahlen␣gegeben");
}

int lese()
{
  int anzFest=0;
  int i, j, z;
```

```
  for(i=0; i<9; i++)   // überprüfe alle Zahlen
    for(j=0; j<9; j++)
    {
      if(zahlen[j][i]<1 || zahlen[j][i]>9)
        zahlen[j][i]=0;
      else
        anzFest++;
    }
  return anzFest;
}

boolean konsistent()
{
  int i, j, z;
  boolean ok=true;

  for(i=0; i<9; i++)
    for(j=0; j<9; j++)
    {
      z = zahlen[j][i]; // speichere den Wert des betrachteten Feld
      if(z>0)
      {
        zahlen[j][i] = 0; // setze das Feld auf "leer" (für die
           Überprüfung)
        ok = ok && checkZahl(i,j,z);
        zahlen[j][i] = z; // schreibe Zahl wieder ins Feld
      }
    }
  return ok;
}

boolean checkZahl(int x, int y, int testzahl)  // true, falls zahl
   in array an Position (x,y) stehen darf
{
  boolean ok = true;
  int i;

  for(i=0; i<9; i++)
  {
    ok = ok && (testzahl != zahlen[y][i]);  // über Zeilen
    ok = ok && (testzahl != zahlen[i][x]);  // über Spalten
    ok = ok && (testzahl != zahlen[(y/3)*3+(i/3)][(x/3)*3+(i%3)]);
    // Bemerkung: (i%3) ergibt die horizontale Koordinate im 9er-
       Feld
    // (i/3) ergibt die vertikale Koordinate im 9er-Feld
  }
  return ok;
}

boolean loese(int ebene)
{
  int i, j, k;
```

```
int minAnzMoeglichkeiten=10;
int imin=0, jmin=0;
int[] moegl = new int[9];
boolean status=true;

// finde Feld mit der kleinsten Abzahl Möglichkeiten
for(i=0; i<9; i++)
  for(j=0; j<9; j++)
    if(zahlen[j][i]==0)
    {
      k = bestimmeMoeglichkeiten(i,j, moegl);
      if(k<minAnzMoeglichkeiten)
      {
        imin = i;
        jmin = j;
        minAnzMoeglichkeiten = k;
      }
    }
if(minAnzMoeglichkeiten==0)  // geht nicht
  return false;
bestimmeMoeglichkeiten(imin,jmin, moegl); // um für das Feld mit
   der Anzahl Möglichkeiten diese zu erhalten
if(ebene==80)  // fertig und letzte Zahl kann gesetzt werden
{
  zahlen[jmin][imin] = moegl[0];
  anzLoesungen++;
  println("Lösung␣"+anzLoesungen);
  for(i=0; i<9; i++)  // Ausgabe
  {
    if(i%3 == 0)
      println("-------------------------");
    for(j=0; j<9; j++)
    {
      if(j%3 == 0)
        print("|␣");
      print(zahlen[i][j]+"␣");
    }
    print("|"+'\n');
  }
  println("-------------------------");
  zahlen[jmin][imin] = 0;
  if(anzLoesungen<N+1)
    return true;
  else
    return false; // zu viele Lösungen
}
if(anzLoesungen<N+1)
{
  for(i=0;i<minAnzMoeglichkeiten; i++)  // Schleife über
     Möglichkeiten und rekursiver Aufruf
  {
    zahlen[jmin][imin] = moegl[i];
```

```
      status = loese(ebene+1) && status;
      zahlen[jmin][imin] = 0;
    }
  }
  return status;
}


int bestimmeMoeglichkeiten(int itest, int jtest, int[] moegl)
{
  boolean ok;
  int j, z;
  int[] anzahl = new int[10];

  for(j=1; j<10; j++)
    anzahl[j]=0;
  for(j=0; j<9; j++)
  {
    anzahl[zahlen[j][itest]]++;    // gehe über Zeile itest
    anzahl[zahlen[jtest][j]]++;   // und über Spalte jtest
    anzahl[zahlen[(jtest/3)*3+(j%3)][(itest/3)*3+j/3]]++;   // und
       über das 9er-Feld
  }
  z=0;
  for(j=1; j<10; j++)
    if(anzahl[j]==0)
      moegl[z++]=j;
  return z;
}
```

Es gibt genau eine Lösung. (Das Sudoku ist übrigens als "schwer" klassifiziert.)

5	8	3	1	7	4	6	2	9
2	1	4	5	6	9	7	8	3
9	7	6	2	8	3	4	5	1
4	5	7	8	3	1	9	6	2
3	6	9	4	2	5	8	1	7
8	2	1	9	6	7	5	3	4
7	9	8	3	1	6	2	4	5
6	3	5	7	4	2	1	9	8
1	4	2	9	5	8	3	7	6