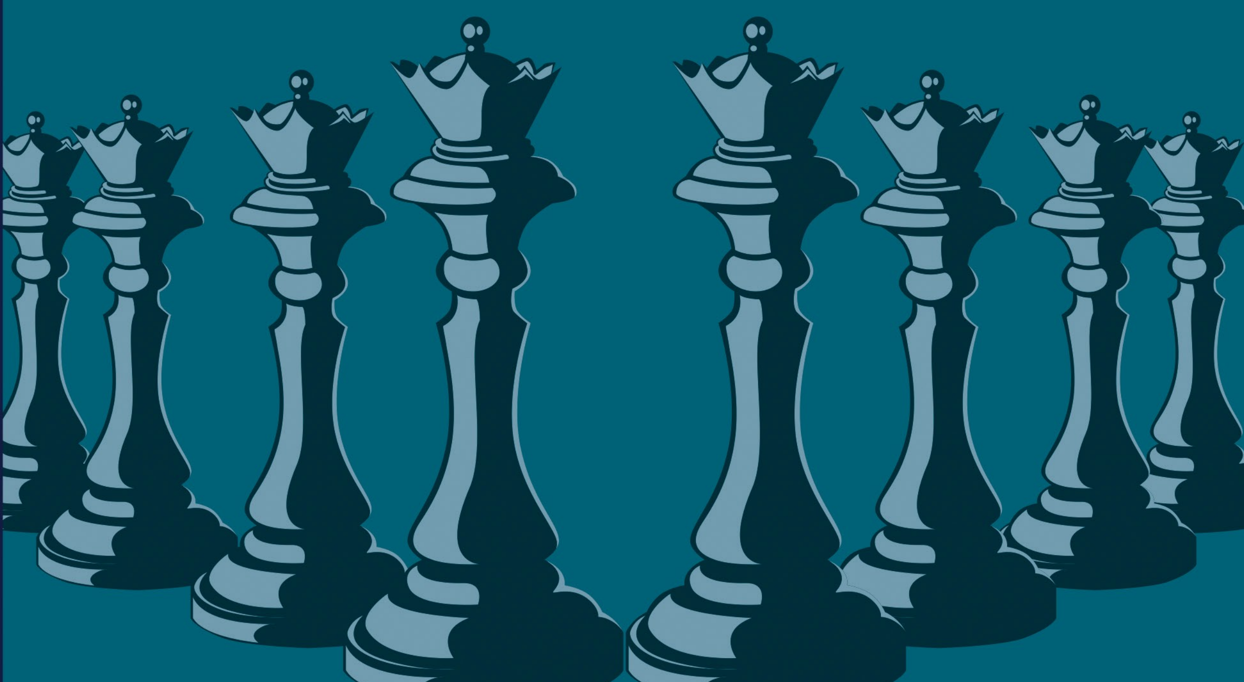
David Kopec

Der Feinschliff für Ihre Programmier-Skills

Algorithmen in Python

32 Klassiker vom Damenproblem bis zu Neuronalen Netzen



Alle Beispielprojekte zum Download

Rheinwerk Computing

Einleitung

Danke, dass Sie *Algorithmen in Python* gekauft haben. Python ist eine der populärsten Programmiersprachen der Welt, und Menschen mit verschiedensten Hintergründen werden Python-Programmierer. Einige haben eine formale Informatikausbildung. Andere lernen Python als Hobby. Wieder andere benutzen Python in ihrem beruflichen Umfeld, obwohl Softwareentwicklung nicht ihre Hauptaufgabe ist. Die Aufgaben in diesem Buch haben einen mittleren Schwierigkeitsgrad und helfen so erfahrenen Programmierern, Ideen aus ihrer Informatikausbildung aufzufrischen und dabei einige fortgeschrittene Features der Sprache zu erlernen. Programmierer im Selbststudium werden ihre Informatikausbildung beschleunigen, indem sie lernen, klassische Aufgaben in der Sprache ihrer Wahl zu lösen: Python. Dieses Buch behandelt eine so große Bandbreite von Problemlösungsstrategien, dass wirklich für jeden etwas dabei ist.

Dieses Buch ist keine Einführung in Python. Es gibt diverse hervorragende Bücher von Manning, dem Rheinwerk Verlag und anderen Verlagen, die diese Aufgabe erfüllen.[1] Wenngleich dieses Buch Python 3.7 voraussetzt, wird die Beherrschung jeder Facette der neuesten Version von Python nicht vorausgesetzt. Tatsächlich wurde das Buch mit dem Anspruch geschrieben, als Lernmaterial zu dienen, das Lesern hilft, genau diese Beherrschung zu erreichen. Andererseits ist dieses Buch nicht für Leser geeignet, denen Python vollkommen neu ist.

Warum Python?

Python wird in so unterschiedlichen Aufgabenfeldern wie Datenwissenschaft, Filmproduktion, Informatikausbildung, IT-Management und vielen anderen eingesetzt. Praktisch kein Gebiet der Computeranwendung ist von Python unberührt geblieben (außer vielleicht die Kernel-Entwicklung). Python ist beliebt wegen seiner Flexibilität, seiner schönen und knappen Syntax, seiner objektorientierten Reinheit und seiner lebendigen Community. Die starke Community ist wichtig, weil sie dafür sorgt, dass Python Neueinsteiger willkommen heißt und ein umfangreiches Ökosystem verfügbarer Bibliotheken aufweist, auf die Entwickler aufsetzen können.

1 Wenn Sie Ihre Reise durch Python gerade erst beginnen, sollten Sie sich zuerst The Quick Python Book, 3. Auflage, von Naomi Ceder (Manning, 2018) anschauen, bevor Sie mit diesem Buch anfangen. [deutschsprachige Alternative: »Einstieg in Python«, 6. Auflage, von Thomas Theis, Rheinwerk Verlag 2020]

Aus den vorgenannten Gründen wird Python manchmal als anfängerfreundliche Sprache betrachtet, und diese Charakterisierung ist wahrscheinlich korrekt. Die meisten Leute würden beispielsweise zustimmen, dass Python leichter zu lernen ist als C++, und es ist so gut wie sicher, dass seine Community netter zu Einsteigern ist. In der Folge lernen viele Leute Python, weil es zugänglich ist, und sie beginnen sehr schnell damit, die Programme zu schreiben, die sie schreiben möchten. Aber sie haben womöglich nie eine Informatikausbildung erhalten, in der ihnen die ganze Vielfalt verfügbarer, leistungsfähiger Problemlösungsstrategien beigebracht wurde. Wenn Sie zu den Programmierern gehören, die mit Python, aber nicht mit der Informatik vertraut sind, ist dieses Buch etwas für Sie.

Andere lernen Python als zweite, dritte, vierte oder fünfte Sprache, nachdem sie lange in der Softwareentwicklung gearbeitet haben. Bei ihnen wird es den Python-Lernfortschritt beschleunigen, wenn sie alte Aufgaben wiedersehen, mit denen sie bereits in anderen Sprachen zu tun hatten. Für sie kann dieses Buch der Wiederauffrischung vor einem Vorstellungsgespräch dienen oder sie mit Problemlösungsstrategien vertraut machen, deren Einsatz sie bei ihrer Arbeit zuvor nicht in Betracht gezogen haben. Ich würde ihnen raten, das Inhaltsverzeichnis zu überfliegen, um herauszufinden, ob es in diesem Buch Themen gibt, die sie begeistern.

Was ist eine klassische Informatikaufgabe?

Manche sagen, dass sich Computer zur Informatik verhalten wie Teleskope zur Astronomie. Wenn das der Fall ist, dann entspricht eine Programmiersprache vielleicht einer Teleskoplinse. In jedem Fall wird der Begriff »klassische Informatikaufgaben« hier für »Programmieraufgaben, die typischerweise im Grundstudium der Informatik gelehrt werden« verwendet.

Es gibt bestimmte Programmieraufgaben, die neuen Programmierern zum Lösen vorgelegt werden und verbreitet genug sind, um als Klassiker bezeichnet zu werden, ob im Rahmen des Unterrichts während eines Bachelor-Studiengangs (in Informatik, Software-Engineering und ähnlichen) oder in einem Programmierlehrbuch mittlerer Lernstufe (zum Beispiel einem Einführungsbuch über künstliche Intelligenz oder Algorithmen). Eine Auswahl solcher Aufgaben finden Sie in diesem Buch.

Die Aufgaben reichen von trivialen, die mit ein paar Zeilen Code gelöst werden können, bis hin zu komplexen, die den Aufbau von Systemen über mehrere Kapitel hinweg benötigen. Einige Aufgaben berühren die künstliche Intelligenz, während andere bloß etwas Nachdenken erfordern. Einige Aufgaben sind praktisch, andere eher kurios.

Welche Arten von Aufgaben gibt es in diesem Buch?

Kapitel 1 stellt Problemlösungsstrategien vor, die den meisten Leserinnen und Lesern vertraut vorkommen werden. Memoisation und Bit-Manipulation sind grundlegende Bausteine anderer Verfahren, die in späteren Kapiteln betrachtet werden.

Auf diese behutsame Einführung folgt Kapitel 2, das sich mit Suchaufgaben beschäftigt. Die Suche ist ein so umfangreiches Thema, dass man vermutlich die meisten Aufgaben in diesem Buch unter dieser Überschrift zusammenfassen könnte. Kapitel 2 stellt die grundlegenden Suchalgorithmen vor, darunter binäre Suche, Tiefensuche, Breitensuche und A*. Diese Algorithmen werden im Rest des Buchs wiederverwendet.

In Kapitel 3 schreiben Sie ein Framework zur Lösung einer breiten Palette von Aufgaben, die sich abstrakt durch Variablen mit beschränktem Wertebereich und zwischen ihnen geltenden Bedingungen beschreiben lassen. Dazu gehören Klassiker wie das Acht-Damen-Problem, das Einfärben der Landkarte Australiens oder das kryptoarithmetische Rätsel SEND+MORE=MONEY.

Kapitel 4 erforscht die Welt der Graphenalgorithmen, deren Anwendungsgebiete für Uneingeweihte überraschend vielfältig erscheinen. In diesem Kapitel schreiben Sie eine Graphen-Datenstruktur und verwenden sie zur Lösung diverser klassischer Optimierungsaufgaben.

Kapitel 5 beschäftigt sich mit genetischen Algorithmen, einem Verfahren, das weniger deterministisch ist als die meisten in diesem Buch behandelten, aber manchmal Probleme lösen kann, die traditionelle Algorithmen nicht innerhalb einer vernünftigen Zeitspanne lösen können.

Kapitel 6 behandelt k-Means-Cluster-Algorithmen und ist vielleicht das algorithmenspezifischste Kapitel im Buch. Diese Clustertechnik ist einfach zu implementieren, einfach zu verstehen und auf vielen Gebieten anwendbar.

Kapitel 7 versucht zu erklären, was ein neuronales Netzwerk ist, und gibt Lesern einen Einblick in den Aufbau eines sehr einfachen neuronalen Netzwerks. Das Kapitel strebt nicht danach, dieses faszinierende und sich stetig weiterentwickelnde Wissensgebiet umfassend zu behandeln. In diesem Kapitel schreiben Sie ein neuronales Netzwerk nach einfachen Prinzipien und ohne externe Bibliotheken, damit Sie genau sehen können, wie ein neuronales Netzwerk arbeitet.

In Kapitel 8 geht es um Adversarial Search in Zwei-Spieler-Spielen mit perfekter Information. Sie lernen einen Suchalgorithmus kennen, der Minimax genannt wird und verwendet werden kann, um einen künstlichen Gegner zu entwickeln, der Spiele wie Schach, Dame und Vier Gewinnt gut spielen kann.

Kapitel 9 behandelt schließlich interessante (und Spaß machende) Aufgaben, die nirgendwo anders ins Buch passten.

Für wen ist dieses Buch geeignet?

Dieses Buch eignet sich sowohl für fortgeschrittene als auch für erfahrene Programmierer. Erfahrene Programmierer, die ihre Python-Kenntnisse vertiefen möchten, finden hier hinreichend bekannte Aufgaben aus ihrer Informatik- oder Programmierausbildung. Fortgeschrittene Programmierer lernen diese klassischen Aufgaben in der Sprache ihrer Wahl kennen: Python. Entwickler, die sich auf Coding-Vorstellungsgespräche vorbereiten, finden in diesem Buch hilfreiches Vorbereitungsmaterial.

Neben professionellen Programmierern werden auch Studenten im Grundstudium der Informatik dieses Buch vermutlich hilfreich finden. Es unternimmt jedoch keinen Versuch, eine gründliche Einführung in Datenstrukturen und Algorithmen zu sein. *Dies ist kein Lehrbuch über Datenstrukturen und Algorithmen*. Sie werden auf seinen Seiten keine Beweise oder ausführliche Big-O-Notation finden. Stattdessen handelt es sich um ein zugängliches, praktisches Tutorial für Problemlösungsverfahren, die das Endergebnis von Kursen über Datenstrukturen, Algorithmen und künstliche Intelligenz sein sollten.

Noch einmal: Kenntnisse der Syntax und Semantik von Python werden vorausgesetzt. Einem Leser, der keinerlei Programmiererfahrung hat, bringt dieses Buch wenig, und ein Programmierer ohne jegliche Python-Erfahrung wird sich fast sicher schwertun. *Algorithmen in Python* ist mit anderen Worten ein Buch für aktive Python-Programmierer und Informatikstudenten.

Python-Version, Quellcode-Repository und Type-Hints

Der Quellcode in diesem Buch wurde in Übereinstimmung mit Version 3.7 der Sprache Python geschrieben. Er verwendet Features von Python, die erst seit Python 3.7 verfügbar sind, so dass ein Teil des Codes nicht mit älteren Versionen von Python funktionieren wird. Anstatt sich abzumühen, die Beispiele in einer älteren Version zum Laufen zu bringen, laden Sie bitte einfach die neueste Version von Python herunter, bevor Sie mit dem Buch beginnen.[2]

2 [Anmerkung des Übersetzers: Zum Zeitpunkt der Übersetzung ist die aktuelle Version 3.8, und alle Beispiele funktionieren damit tadellos.]

Dieses Buch verwendet ausschließlich die Python-Standardbibliothek (mit einer kleinen Ausnahme in Kapitel 2, wo das Modul `typing_extensions` installiert wird), so dass sämtlicher Code in diesem Buch auf jeder Plattform laufen sollte, auf der Python unterstützt wird (macOS, Windows, GNU/Linux und so weiter). Der Code in diesem Buch wurde ausschließlich mit CPython getestet (dem von *python.org* verfügbaren Haupt-Python-Interpreter), wird aber wahrscheinlich größtenteils auch in einer Python-3.7-kompatiblen Version eines anderen Python-Interpreters laufen.

Dieses Buch erklärt nicht, wie Sie Python-Tools wie Editoren, IDEs, Debugger oder das Python-REPL verwenden. Der Quellcode des Buches ist online im GitHub-Repository verfügbar: *https://github.com/davecom/ClassicComputerScienceProblemsInPython/*.[3] Der Quellcode ist in Ordnern nach Kapiteln organisiert. Wenn Sie jedes Kapitel lesen, finden Sie den Namen einer Quelldatei in der Überschrift jedes Codelistings. Diese Quelldatei finden Sie in ihrem jeweiligen Ordner im Repository. Sie sollten das Programm ausführen können, indem Sie `python3 filename.py` oder `python filename.py` eingeben, je nach Konfiguration Ihres Computers, was den Namen des Python-3-Interpreters angeht.

Jedes Codelisting in diesem Buch macht Gebrauch von Python-Type-Hints, auch *Typ-Annotationen* genannt. Diese Annotationen sind ein relativ neues Feature für die Sprache Python, und sie mögen auf Python-Programmierer, die sie noch nie zuvor gesehen haben, einschüchternd wirken. Sie werden aus drei Gründen verwendet:

1. Sie bieten Klarheit über die Typen von Variablen, Funktionsparametern und Funktionsrückgabewerten.
2. Als Konsequenz aus Grund 1 dokumentieren sie in gewisser Weise automatisch den Code. Anstatt einen Kommentar oder Docstring zu durchsuchen, um den Rückgabewert einer Funktion zu finden, können Sie einfach ihre Signatur betrachten.
3. Sie ermöglichen es, den Code auf Korrektheit der Datentypen zu prüfen. Ein populärer Python-Typchecker ist mypy.

Nicht jeder ist ein Fan von Type-Hints, und sie im gesamten Buch einzusetzen, war in gewisser Weise ein Wagnis. Ich hoffe, dass sie eine Hilfe und keine Hürde darstellen. Es benötigt etwas mehr Zeit, Python mit Type-Hints zu schreiben, bietet aber mehr Klarheit beim späteren Lesen. Ein interessanter Fakt ist, dass Type-Hints keine Auswirkungen auf die Ausführung des Codes im Python-Interpreter haben. Sie können die Type-Hints aus jedem Code in diesem Buch entfernen, und er sollte immer noch laufen. Wenn Sie noch nie zuvor Type-Hints gesehen haben und glauben, dass Sie eine gründlichere Einfüh-

3 [Der Rheinwerk Verlag stellt den Quellcode auf der Webseite zum Buch unter »Materialien zum Buch« zum Download bereit: *https://www.rheinwerk-verlag.de/5143*]

rung benötigen, bevor Sie sich das Buch vornehmen, lesen Sie bitte Anhang C, der einen Crashkurs in Type-Hints liefert.

Keine Grafik, kein UI-Code, nur die Standardbibliothek

Es gibt in diesem Buch keine Beispiele, die grafische Ausgaben erzeugen oder Gebrauch von einer grafischen Benutzeroberfläche (GUI) machen. Warum? Das Ziel ist, die gestellten Aufgaben durch Code zu lösen, der so kompakt und lesbar wie möglich ist. Oft steht Grafik diesem Ziel im Weg oder macht Lösungen deutlich komplexer, als sie zur Veranschaulichung des jeweiligen Verfahrens oder Algorithmus sein müssten.

Außerdem macht die Entscheidung, keinen Gebrauch von einem GUI-Framework zu machen, jeglichen Code in diesem Buch äußerst gut portierbar. Er kann ebenso gut auf einer Embedded-Distribution von Python unter Linux laufen wie auf einem Desktop unter Windows. Außerdem wurde die bewusste Entscheidung getroffen, nur Pakete aus der Python-Standardbibliothek zu verwenden statt externer Bibliotheken, anders als in den meisten Python-Büchern für Fortgeschrittene. Warum? Das Ziel ist es, Problemlösungsverfahren von ihren Grundprinzipien aus zu lehren, nicht »pip install Lösung« anzubieten. Indem Sie jede Aufgabe von Grund auf durcharbeiten müssen, gewinnen Sie hoffentlich ein Verständnis dafür, wie populäre Bibliotheken hinter den Kulissen arbeiten. In jedem Fall macht die ausschließliche Verwendung der Standardbibliothek den Code in diesem Buch portierbarer und einfacher ausführbar.

Das soll nicht heißen, dass grafische Lösungen einen Algorithmus nicht manchmal besser veranschaulichen können als textbasierte Lösungen. Das war einfach nicht der Fokus dieses Buches. Es würde eine zusätzliche Ebene unnötiger Komplexität hinzufügen.

Teil einer Serie

Dies ist das zweite Buch einer Serie namens *Classic Computer Science Problems*, die von Manning verlegt wird. Das erste Buch war *Classic Computer Science Problems in Swift*, erschienen 2018. In jedem Buch der Serie versuchen wir, sprachspezifische Besonderheiten herauszustellen, während wir (ungefähr) dieselben Informatikaufgaben lehren.

Wenn Ihnen dieses Buch gefällt und Sie beschließen, eine andere in dieser Serie behandelte Sprache zu erlernen, finden Sie auf dem Weg von einem Buch zum nächsten möglicherweise einen einfachen Weg, Ihre Kenntnisse in dieser Sprache zu vertiefen. Im Moment umfasst die Serie nur Swift und Python. Ich habe die ersten beiden Bücher selbst geschrieben, weil ich über große Erfahrung mit beiden Sprachen verfüge, aber wir

diskutieren bereits Pläne für weitere Bücher in der Reihe mit Experten in anderen Sprachen, die als Co-Autoren in Frage kommen. Wenn Ihnen dieses Buch gefällt, ermutige ich Sie, danach Ausschau zu halten. Weitere Informationen über die Serie finden Sie unter *https://classicproblems.com/*.

Kapitel 2
Suchaufgaben

»Suche« ist ein so breit gefächerter Begriff, dass dieses ganze Buch *Klassische Suchaufgaben in Python* heißen könnte. In diesem Kapitel geht es um Kernalgorithmen für die Suche, die jeder Programmierer kennen sollte. Trotz des deklaratorischen Titels erhebt es keinen Anspruch auf Vollständigkeit.

2.1 DNA-Suche

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

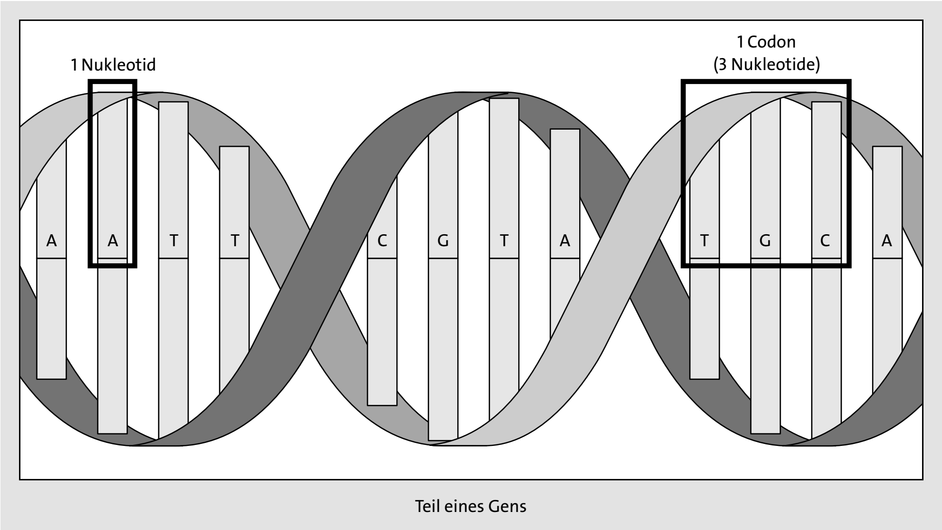


Abbildung 2.1 Ein Nukleotid wird durch einen der Buchstaben A, C, G oder T dargestellt. Ein Codon besteht aus drei Nukleotiden und ein Gen aus mehreren Codons.

2.1.1 DNA speichern

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

```
from enum import IntEnum
from typing import Tuple, List
Nucleotide: IntEnum = IntEnum('Nucleotide', ('A', 'C', 'G', 'T'))
```

Listing 2.1 dna_search.py

`Nucleotide` hat den Typ `IntEnum` statt eines einfachen `Enum`, weil `IntEnum` »kostenlose« Vergleichsoperatoren (<, >= und so weiter) mitliefert. Diese Operatoren in einem Datentyp zu haben, ist notwendig, damit die Suchalgorithmen, die wir implementieren werden, mit ihnen arbeiten können. `Tuple` und `List` werden aus dem Paket `typing` importiert, um mit Type-Hints auszuhelfen.

Codons lassen sich als Tupel von drei `Nucleotide`-Objekten definieren. Ein Gen wiederum ist als Liste von `Codon`-Objekten definiert.

```
Codon = Tuple[Nucleotide, Nucleotide, Nucleotide]  # Type-Alias für Codons
Gene = List[Codon]  # Typ-Alias für Gene
```

Listing 2.2 dna_search.py (Fortsetzung)

Hinweis

Auch wenn wir später `Codon`-Objekte miteinander vergleichen müssen, brauchen wir keine spezifische Klasse zu definieren, in der der Operator < explizit für `Codon` implementiert ist. Das liegt daran, dass Python eingebaute Unterstützung für Vergleiche zwischen Tupeln aus Typen besitzt, die ihrerseits vergleichbar sind.

Gene werden im Internet typischerweise ein Dateiformat haben, das einen riesigen String mit sämtlichen Nukleotiden in der Sequenz des Gens enthält. Wir definieren einen solchen String für ein imaginäres Gen und nennen ihn `gene_str`.

```
gene_str: str = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

Listing 2.3 dna_search.py (Fortsetzung)

Außerdem brauchen wir eine Hilfsfunktion, die einen `str` in ein `Gene` umwandelt.

```
def string_to_gene(s: str) -> Gene:
    gene: Gene = []
    for i in range(0, len(s), 3):
```



```
        if (i + 2) >= len(s):  # Nicht über das Ende hinausschießen!
            return gene
        #  Codon aus drei Nukleotiden initialisieren
        codon: Codon = (Nucleotide[s[i]], Nucleotide[s[i + 1]],
         Nucleotide[s[i + 2]])
        gene.append(codon)  # Codon zum Gen hinzufügen
    return gene
```

Listing 2.4 dna_search.py (Fortsetzung)

`string_to_gene()` durchwandert kontinuierlich den übergebenen `str` und konvertiert seine jeweils nächsten drei Zeichen in `Codon`s, die am Ende eines neuen `Gene`-Objekts hinzugefügt werden. Wenn die Funktion merkt, dass sich zwei Positionen von der aktuellen entfernt kein `Nucleotide` mehr befindet (siehe `if`-Anweisung innerhalb der Schleife), dann weiß sie, dass sie das Ende eines unvollständigen Gens erreicht hat und überspringt die restlichen ein bis zwei Nukleotide.

`string_to_gene()` kann verwendet werden, um den `str` `gene_str` in ein `Gene` umzuwandeln.

```
my_gene: Gene = string_to_gene(gene_str)
```

Listing 2.5 dna_search.py (Fortsetzung)

2.1.2 Lineare Suche

Eine grundlegende Operation, die wir auf einem Gen ausführen wollen, besteht darin, nach einem bestimmten Codon zu suchen. Das Ziel ist, einfach herauszufinden, ob das Codon innerhalb des Gens existiert oder nicht.

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

Es ist trivial, eine Funktion zu definieren, die eine lineare Suche durchführt. Sie muss einfach jedes Element in einer Datenstruktur durchgehen und auf Gleichheit mit dem gesuchten Element überprüfen. Der folgende Code definiert eine solche Funktion für ein `Gene` und ein `Codon` und probiert sie dann mit `my_gene` und `Codon`s namens `acg` und `gat` aus.

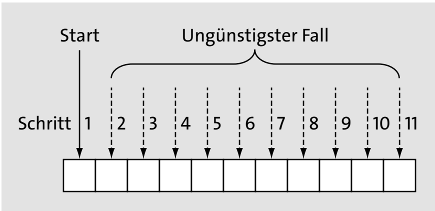


Abbildung 2.2 Im ungünstigsten Fall einer linearen Suche müssen Sie nacheinander jedes Element des Arrays betrachten.

```
def linear_contains(gene: Gene, key_codon: Codon) -> bool:
    for codon in gene:
        if codon == key_codon:
            return True
    return False
acg: Codon = (Nucleotide.A, Nucleotide.C, Nucleotide.G)
gat: Codon = (Nucleotide.G, Nucleotide.A, Nucleotide.T)
print(linear_contains(my_gene, acg))  # True
print(linear_contains(my_gene, gat))  # False
```

Listing 2.6 dna_search.py (Fortsetzung)

Hinweis

Diese Funktion dient lediglich illustrativen Zwecken. Die in Python eingebauten Sequenztypen (`list`, `tuple`, `range`) implementieren alle die Methode `__contains__()`, die es uns erlaubt, mithilfe des Operators `in` nach einem bestimmten Element in ihnen zu suchen. Tatsächlich kann der Operator mit jedem Typ verwendet werden, der `__contains__()` implementiert. Beispielsweise könnten wir `my_gene` nach `acg` durchsuchen und das Ergebnis ausgeben, indem wir `print(acg in my_gene)` schreiben.

2.1.3 Binärsuche

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

Eine Binärsuche funktioniert, indem sie das mittlere Element einer sortierten Abfolge von Elementen betrachtet, es mit dem gesuchten Element vergleicht, den Suchbereich

aufgrund dieses Vergleichs verkleinert und den Prozess erneut startet. Schauen wir uns ein konkretes Beispiel an.

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

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

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

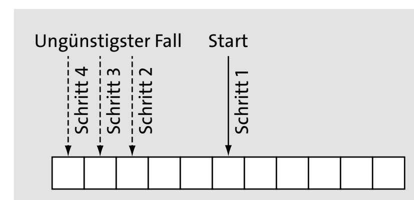


Abbildung 2.3 Im ungünstigsten Fall einer Binärsuche durchsuchen Sie nur lg(n) Elemente der Liste.

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

Eine Binärsuchfunktion für ein Gen und ein Codon unterscheidet sich nicht von einer für jede andere Art von Daten, weil der Typ `Codon` anderen Typen ähnelt und der Typ `Gene` nichts weiter als eine `list` ist.

```
def binary_contains(gene: Gene, key_codon: Codon) -> bool:
    low: int = 0
    high: int = len(gene) - 1
    while low <= high:  # solange es noch einen Suchraum gibt
        mid: int = (low + high) // 2
        if gene[mid] < key_codon:
            low = mid + 1
        elif gene[mid] > key_codon:
            high = mid - 1
        else:
            return True
    return False
```

Listing 2.7 dna_search.py (Fortsetzung)

Schauen wir uns diese Funktion Zeile für Zeile an.

```
low: int = 0
high: int = len(gene) - 1
```

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

```
while low <= high:
```

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

```
mid: int = (low + high) // 2
```

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

```
if gene[mid] < key_codon:
    low = mid + 1
```

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


```
elif gene[mid] > key_codon:
    high = mid - 1
```

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

```
else:
    return True
```

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

Wir können versuchen, unsere Funktion mit demselben Gen und demselben Codon auszuführen, aber wir müssen daran denken, zuerst zu sortieren.

```
my_sorted_gene: Gene = sorted(my_gene)
print(binary_contains(my_sorted_gene, acg))  # True
print(binary_contains(my_sorted_gene, gat))  # False
```

Listing 2.8 dna_search.py (Fortsetzung)

Hinweis

Mithilfe des Moduls `bisect` aus der Python-Standardbibliothek können Sie eine performante Binärsuche bauen: *https://docs.python.org/3/library/bisect.html*.

2.1.4 Ein generisches Beispiel

Die Funktionen `linear_contains()` und `binary_contains()` lassen sich so verallgemeinern, dass sie mit so gut wie jeder Python-Folge arbeiten können. Die folgenden verallgemeinerten Versionen sind beinahe identisch mit denjenigen, die Sie bereits gesehen haben, nur einige Namen und Type-Hints wurden ausgetauscht.

Hinweis

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

Hinweis

Bevor Sie mit dem Buch fortfahren, müssen Sie das Modul `typing_extensions` installieren, indem Sie entweder `pip install typing_extensions` oder `pip3 install typing_extensions` eingeben, je nachdem, wie Ihr Python-Interpreter konfiguriert ist. Sie brauchen dieses Modul, um auf den Typ `Protocol` zuzugreifen, der in einer zukünftigen Version von Python Teil der Standardbibliothek wird (spezifiziert in PEP 544). In einer späteren Version von Python sollte das Modul `typing_extensions` also unnötig werden, und Sie werden in der Lage sein, `from typing import Protocol` statt `from typing_extensions import Protocol` zu schreiben.

```
from __future__ import annotations
from typing import TypeVar, Iterable, Sequence, Generic, List, Callable, Set,
 Deque, Dict, Any, Optional
from typing_extensions import Protocol
from heapq import heappush, heappop

T = TypeVar('T')

def linear_contains(iterable: Iterable[T], key: T) -> bool:
    for item in iterable:
        if item == key:
            return True
    return False

C = TypeVar("C", bound="Comparable")

class Comparable(Protocol):
    def __eq__(self, other: Any) -> bool:
        ...
    def __lt__(self: C, other: C) -> bool:
        ...
    def __gt__(self: C, other: C) -> bool:
        return (not self < other) and self != other

    def __le__(self: C, other: C) -> bool:
        return self < other or self == other
    def __ge__(self: C, other: C) -> bool:
        return not self < other
```



```
def binary_contains(sequence: Sequence[C], key: C) -> bool:
    low: int = 0
    high: int = len(sequence) - 1
    while low <= high:  # Solange es noch einen Suchraum gibt
        mid: int = (low + high) // 2
        if sequence[mid] < key:
            low = mid + 1
        elif sequence[mid] > key:
            high = mid - 1
        else:
            return True
    return False

if __name__ == "__main__":
    print(linear_contains([1, 5, 15, 15, 15, 15, 20], 5))  # True
    print(binary_contains(["a", "d", "e", "f", "z"], "f"))  # True
    print(binary_contains(["john", "mark", "ronald", "sarah"], "sheila"))
     # False
```

Listing 2.9 generic_search.py

Nun können Sie versuchen, nach anderen Datentypen zu suchen. Diese Funktionen können für fast jede Python-Collection verwendet werden. Darin liegt die Macht generisch geschriebenen Codes. Der einzige etwas unglückliche Umstand dieses Beispiels sind die Verrenkungen, die für Pythons Type-Hints in Form der Klasse `Comparable` gemacht werden mussten. Ein `Comparable`-Typ ist ein Typ, der die Vergleichsoperatoren (`<`, `>=` und so weiter) implementiert. In zukünftigen Python-Versionen sollte es eine lesbarere Möglichkeit geben, einen Type-Hint für Typen zu erstellen, die diese gängigen Operatoren implementieren.

2.2 Labyrinthe lösen

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

Unser Labyrinth sei ein zweidimensionales Gitter aus `Cell`-Objekten. Eine `Cell` ist ein `enum` mit `str`-Werten, in dem `" "` einen leeren Platz und `"X"` einen besetzten Platz darstellt. Es gibt noch weitere Fälle, die bei der Ausgabe eines Labyrinths zu Darstellungszwecken verwendet werden.

```
from enum import Enum
from typing import List, NamedTuple, Callable, Optional
import random
from math import sqrt
from generic_search import dfs, bfs, node_to_path, astar, Node
class Cell(str, Enum):
    EMPTY = " "
    BLOCKED = "X"
    START = "S"
    GOAL = "G"
    PATH = "*"
```

Listing 2.10 maze.py

Wieder erledigen wir zunächst eine große Menge von Importen. Beachten Sie, dass der letzte Import (aus `generic_search`) Symbole betrifft, die wir noch nicht definiert haben. Er wurde hier aus Bequemlichkeitsgründen mitgeliefert, aber Sie sollten ihn auskommentieren, bis Sie ihn brauchen.

Wir brauchen eine Möglichkeit, eine einzelne Stelle im Labyrinth anzugeben. Es handelt sich um ein einfaches `NamedTuple` mit Eigenschaften, die Zeile und Spalte der entsprechenden Position darstellen.

```
class MazeLocation(NamedTuple):
    row: int
    column: int
```

Listing 2.11 maze.py (Fortsetzung)

2.2.1 Ein Zufallslabyrinth erzeugen

Unsere Klasse `Maze` speichert intern ein Gitter (eine Liste von Listen), die den Zustand des Labyrinths darstellt. Sie enthält außerdem Instanzvariablen für die Anzahl der Zeilen, die Anzahl der Spalten, den Startort und den Zielort. Ihr Gitter wird zufällig mit besetzten Zellen gefüllt.

Das generierte Labyrinth sollte einigermaßen spärlich besetzt sein, damit es fast immer einen Pfad von einer gegebenen Startposition zu einer gegebenen Zielposition gibt. (Schließlich geht es darum, unseren Algorithmus zu testen.) Wir lassen Aufrufer eines neuen Labyrinths selbst entscheiden, wie spärlich, geben aber einen Standardwert von 20 % besetzten Zellen vor. Wenn eine Zufallszahl die Schwelle des angegebenen `sparseness`-Parameters überschreitet, ersetzen wir einfach einen freien Platz durch eine Wand.

Wenn wir dies für jede mögliche Stele im Labyrinth tun, nähert sich der Grad der spärlichen Besetzung dem übergebenen `sparseness`-Parameter an.

```
class Maze:
    def __init__(self, rows: int = 10, columns: int = 10, sparseness: float =
     0.2, start: MazeLocation = MazeLocation(0, 0), goal: MazeLocation =
     MazeLocation(9, 9)) -> None:
        # Grundlegende Instanzvariablen initialisieren
        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # Das Gitter mit leeren Zellen auffüllen
        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)]
         for r in range(rows)]
        # Gitter mit besetzten Zellen bestücken
        self._randomly_fill(rows, columns, sparseness)
        # Start- und Zielpositionen einfügen
        self._grid[start.row][start.column] = Cell.START
        self._grid[goal.row][goal.column] = Cell.GOAL

    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
```

Listing 2.12 maze.py (Fortsetzung)

Nun, da wir ein Labyrinth haben, wollen wir auch eine Möglichkeit haben, es übersichtlich auf der Konsole auszugeben. Wir wollen, dass seine Zeichen nahe beieinanderstehen, damit es wie ein richtiges Labyrinth aussieht.

```
# Eine schön formatierte Version des Labyrinths für die Ausgabe zurückgeben
def __str__(self) -> str:
    output: str = ""
    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"
    return output
```

Listing 2.13 maze.py (Fortsetzung)

Probieren Sie diese Labyrinthfunktionen aus.

```
maze: Maze = Maze()
print(maze)
```

2.2.2 Weitere Labyrinth-Hilfsfunktionen

Es wird später nützlich sein, eine Funktion zur Hand zu haben, die überprüft, ob wir während der Suche unser Ziel erreicht haben. Wir wollen mit anderen Worten überprüfen, ob eine bestimmte `MazeLocation`, die die Suche erreicht hat, das Ziel ist. Wir können eine solche Methode zu `Maze` hinzufügen.

```
def goal_test(self, ml: MazeLocation) -> bool:
    return ml == self.goal
```

Listing 2.14 maze.py (Fortsetzung)

Wie können wir uns durch unsere Labyrinthe bewegen? Sagen wir, wir können uns von jeder Stelle im Labyrinth aus horizontal oder vertikal um je eine Position pro Durchgang bewegen. Mithilfe dieser Kriterien kann eine `successors()`-Funktion die möglichen Nachfolgepositionen von einer gegebenen `Maze-Location` aus finden. Jedoch wird die Funktion `successors()` für jedes `Maze` unterschiedlich sein, weil jedes `Maze` eine andere Größe und andere Wände hat. Deshalb definieren wir die Funktion als Methode von `Maze`.

```
def successors(self, ml: MazeLocation) -> List[MazeLocation]:
    locations: List[MazeLocation] = []
    if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column]
     != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row + 1, ml.column))
    if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row - 1, ml.column))
    if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1]
     != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column + 1))
    if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column - 1))
    return locations
```

Listing 2.15 maze.py (Fortsetzung)

`successors()` prüft einfach die Zellen über, unter, links und rechts von einer `Maze-Location` in einem `Maze`, um zu überprüfen, ob es leere Positionen gibt, zu denen von der aktuellen aus gewechselt werden kann. Die Funktion vermeidet es auch, Positionen jenseits der Grenzen des `Maze` zu prüfen. Sie fügt jede mögliche `MazeLocation`, die sie findet, zu einer Liste hinzu, die schließlich an die aufrufende Stelle zurückgibt.

2.2.3 Tiefensuche

Eine *Tiefensuche* (englisch *depth-first search*, DFS) tut, was ihr Name vermuten lässt: Eine Suche, die so tief wie möglich geht, bevor sie zum letzten Entscheidungspunkt zurückgeht, wenn sie eine Sackgasse erreicht. Wir implementieren eine generische Tiefensuche, die unser Labyrinthproblem lösen kann. Sie ist auch für andere Aufgaben wiederverwendbar. Abbildung 2.4 veranschaulicht eine laufende Tiefensuche in einem Labyrinth.

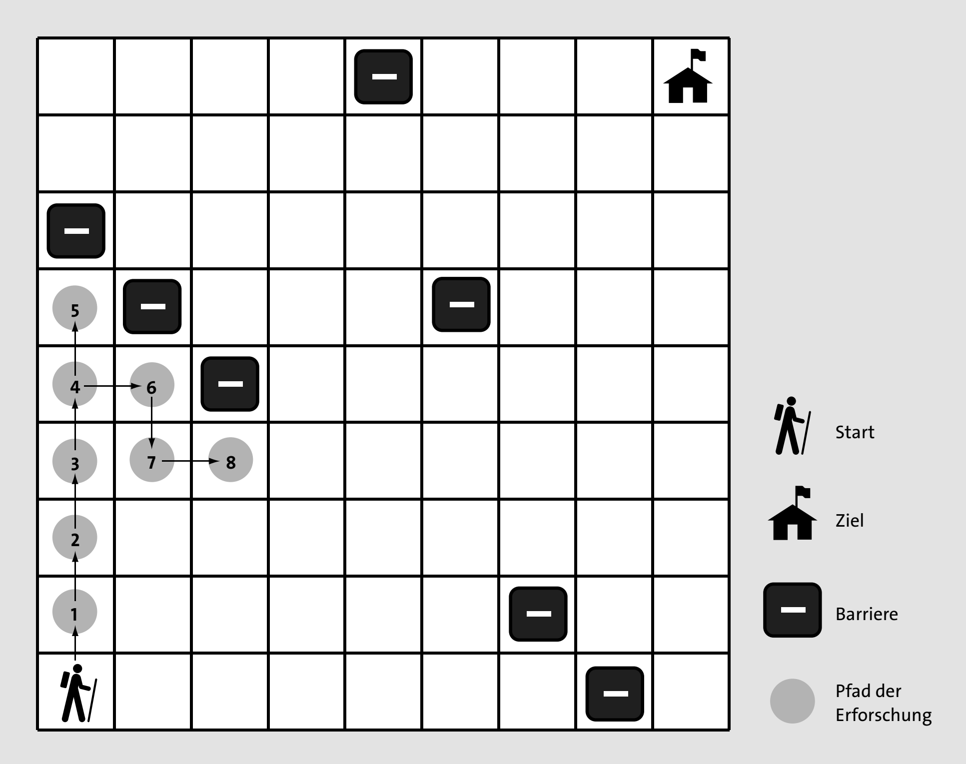


Abbildung 2.4 Bei der Tiefensuche folgt die Suche einem fortlaufend tieferen Pfad, bis sie auf ein Hindernis trifft und bis zum letzten Entscheidungspunkt zurückkehren muss.

Stapel

Die Tiefensuche basiert auf einer Datenstruktur, die als *Stapel* (englisch *stack*) bekannt ist. (Wenn Sie in Kapitel 1 alles Nötige über Stapel gelesen haben, können Sie diesen Abschnitt überspringen.) Ein Stapel ist eine Datenstruktur, die nach dem Last-In-First-Out-Prinzip (LIFO) arbeitet. Stellen Sie sich einen Papierstapel vor. Das letzte Blatt Papier, das oben auf den Stapel gelegt wird, ist das erste, das wieder heruntergenommen wird. Üblicherweise baut ein Stapel auf eine primitivere Datenstruktur wie etwa eine Liste auf. Wir bauen die Implementierung unseres Stapels auf Pythons Typ `list` auf.

Stapel besitzen im Allgemeinen mindestens zwei Operationen:

- `push()` – platziert ein Element als oberstes auf den Stapel.
- `pop()` – entfernt das oberste Element vom Stapel und gibt es zurück.

Wir implementieren beide Methoden sowie die Eigenschaft `empty`, um zu überprüfen, ob der Stapel noch weitere Elemente enthält. Wir fügen den Code für den Stapel zur Datei *generic_search.py* hinzu, mit der wir zuvor in diesem Kapitel gearbeitet haben. Wir haben bereits alle erforderlichen Importe durchgeführt.

```
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []
    @property
    def empty(self) -> bool:
        return not self._container  # not ist für leere Container True

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.pop()  # LIFO

    def __repr__(self) -> str:
        return repr(self._container)
```

Listing 2.16 generic_search.py (Fortsetzung)

Beachten Sie, dass zur Implementierung eines Stapels in Python nichts weiter erforderlich ist, als Elemente immer an seinem rechten Ende hinzuzufügen und stets auch wieder von seinem äußerst rechten Ende zu entfernen. Die Methode `pop()` einer Liste schlägt fehl, wenn die Liste keine weiteren Elemente enthält, also schlägt `pop()` in diesem Fall auch bei einem `Stack` fehl.

Der DFS-Algorithmus

Wir brauchen noch eine weitere Kleinigkeit, bevor wir uns daran machen können, DFS zu implementieren. Wir brauchen eine `Node`-Klasse, mit der wir uns während der Suche merken, wie wir von einem Zustand in den nächsten (oder von einer Position zu einer anderen) gelangt sind. Sie können sich `Node` als Wrapper um einen Zustand vorstellen. Im Fall unserer Labyrinth-Lösungs-Aufgabe sind diese Zustände vom Typ `MazeLocation`. Wir nennen den `Node`, von dem aus wir in den aktuellen Zustand gelangt sind, den `parent` dieses Zustands. Außerdem definieren wir die Eigenschaften `cost` und `heuristic` sowie die Methode `__lt__()` für unsere `Node`-Klasse, damit wir all dies später im A*-Algorithmus wiederverwenden können.

```
class Node(Generic[T]):
    def __init__(self, state: T, parent: Optional[Node], cost: float =
     0.0, heuristic: float = 0.0) -> None:
        self.state: T = state
        self.parent: Optional[Node] = parent
        self.cost: float = cost
        self.heuristic: float = heuristic

    def __lt__(self, other: Node) -> bool:
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

Listing 2.17 generic_search.py (Fortsetzung)

> **Tipp**
> Der Typ `Optional` gibt an, dass ein Wert eines parametrisierten Typs entweder auf eine Variable oder auf `None` verweisen kann.

> **Tipp**
> Die Zeile `from __future__ import annotations` zu Beginn der Datei ermöglicht es `Node`, in den Type-Hints seiner Methoden auf sich selbst zu verweisen. Ohne diese Zeile müssten wir den Type-Hint als String in Anführungszeichen setzen (zum Beispiel `'Node'`). In künftigen Versionen von Python wird es nicht mehr nötig sein, `annotations` zu importieren. Siehe PEP 563, »Postponed Evaluation of Annotations«, für weitere Informationen: *http://mng.bz/pgzR*.

Eine laufende Tiefensuche muss sich zwei Datenstrukturen merken: Den Stapel der Zustände (oder »Positionen«), die wir für die Suche in Betracht ziehen, nennen wir `frontier` (Grenzland), und die Menge der Zustände, die wir bereits durchsucht haben, `explored` (erforscht). Solange es im Grenzland noch weitere Zustände zu besuchen gibt, prüft DFS weiterhin, ob es sich bei ihnen um das Ziel handelt (wenn ein Zustand das Ziel ist, hält DFS an und gibt es zurück), und fügt ihre Nachfolger zum Grenzland hinzu. Zudem wird jeder Zustand, der bereits durchsucht wurde, als erforscht gekennzeichnet, so dass die Suche nicht im Kreis läuft, indem sie zuvor besuchte Zustände als Nachfolger erreicht. Wenn das Grenzland leer ist, bedeutet dies, dass nirgendwo mehr gesucht werden kann.

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
 List[T]]) -> Optional[Node[T]]:
    # frontier bezeichnet, wohin wir noch gehen müssen
    frontier: Stack[Node[T]] = Stack()
    frontier.push(Node(initial, None))
    # explored bezeichnet, wo wir schon waren
    explored: Set[T] = {initial}

    # Weitersuchen, solange es noch etwas zu entdecken gibt
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # Wenn wir das Ziel gefunden haben, sind wir fertig
        if goal_test(current_state):
            return current_node
        # Prüfen, wohin wir als Nächstes gehen können und wo wir noch
        # nicht gesucht haben
        for child in successors(current_state):
            if child in explored:  # Bereits durchsuchte Kindknoten überspringen
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # Alles durchsucht, aber nie das Ziel gefunden
```

Listing 2.18 generic_search.py (Fortsetzung)

Wenn die Funktion `dfs()` erfolgreich ist , gibt sie den `Node` zurück, der den Zielzustand enthält. Der Pfad vom Start bis zum Ziel kann rekonstruiert werden, indem wir von diesem `Node` mithilfe der Eigenschaft `parent` schrittweise zu dessen Vorgängern zurückgehen.

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]
    # Rückwärts vom Ende zum Anfang arbeiten
    while node.parent is not None:
```



```
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

Listing 2.19 generic_search.py (Fortsetzung)

Zu Anzeigezwecken ist es hilfreich, das Labyrinth mit dem erfolgreichen Pfad, dem Anfangs- und dem Zielzustand zu markieren. Es ist auch von Nutzen, einen Pfad entfernen zu können, damit wir unterschiedliche Suchalgorithmen auf dasselbe Labyrinth anwenden können. Die folgenden beiden Methoden sollten zur Klasse `Maze` in *maze.py* hinzugefügt werden.

```
def mark(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.PATH
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
def clear(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
```

Listing 2.20 maze.py (Fortsetzung)

Das war eine lange Reise, aber nun sind wir endlich in der Lage, das Labyrinth zu lösen.

```
if __name__ == "__main__":
    # DFS testen
    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_
test, m.successors)
    if solution1 is None:
        print("Keine Lösung mit Tiefensuche gefunden!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print(m)
        m.clear(path1)
```

Listing 2.21 maze.py (Fortsetzung)

Eine erfolgreiche Lösung wird etwa so aussehen:

```
S****X X
 X  *****
       X*
 XX******X
  X*
  X**X
 X  *****
        *
     X  *X
        *G
```

Die Sternchen repräsentieren den Pfad vom Start zum Ziel, den unsere Tiefensuche-Funktion gefunden hat. Denken Sie daran, dass nicht jedes Labyrinth eine Lösung hat, da jedes Labyrinth per Zufall generiert wird.

2.2.4 Breitensuche

Wie Sie vielleicht merken, erscheinen die von der Tiefensuche gefundenen Lösungspfade für die Labyrinthe unnatürlich. Es handelt sich üblicherweise nicht um die kürzesten Pfade. Die *Breitensuche* (englisch *breadth-first search*, BFS) findet stets den kürzesten Pfad, indem sie in jeder Iteration der Suche systematisch eine Ebene von Knotenpunkten weiter vom Startzustand entfernt durchsucht. Es gibt spezifische Aufgaben, für die die Tiefensuche wahrscheinlich schneller eine Lösung findet als die Breitensuche und umgekehrt. Deshalb ist die Entscheidung zwischen den beiden immer ein Kompromiss zwischen der Möglichkeit, schnell eine Lösung zu finden, und der Gewissheit, den kürzesten Weg zum Ziel zu finden (falls einer existiert). Abbildung 2.5 veranschaulicht die laufende Breitensuche in einem Labyrinth.

Um zu verstehen, warum eine Tiefensuche manchmal schneller ein Ergebnis zurückliefert als eine Breitensuche, stellen Sie sich vor, Sie suchen eine Markierung auf einer bestimmten Schicht einer Zwiebel. Wer die Tiefensuche verwendet, stößt praktisch ein Messer ins Herz der Zwiebel und untersucht willkürlich die herausgeschnittenen Stücke. Sollte sich die markierte Schicht in der Nähe des herausgeschnittenen Stücks befinden, besteht eine Chance, dass er sie schneller findet als jemand, der eine Breitensuche-Strategie verwendet, bei der er die Zwiebel fein säuberlich Schicht für Schicht schält.

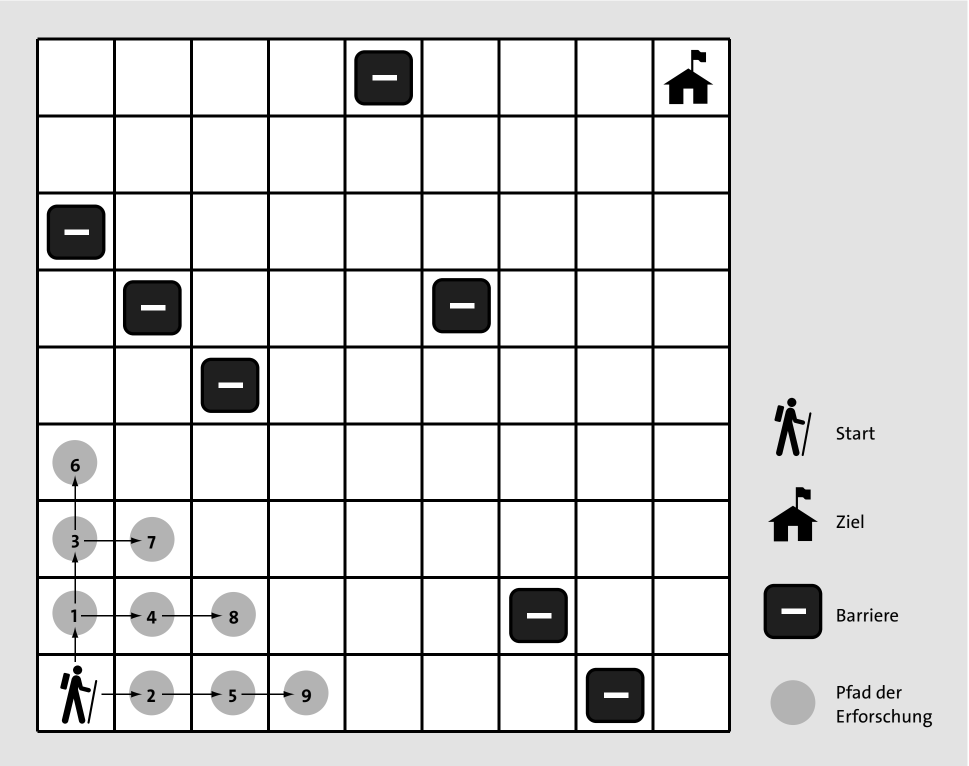


Abbildung 2.5 Bei einer Breitensuche werden die am nächsten an der Startposition befindlichen Elemente zuerst durchsucht.

Um sich ein besseres Bild darüber machen zu können, warum die Breitensuche immer den kürzesten Lösungspfad findet, sofern einer existiert, stellen Sie sich vor, Sie versuchen, die Strecke einer Zugfahrt von Boston nach New York mit der geringsten Anzahl von Haltestellen zu finden. Wenn Sie immer weiter in dieselbe Richtung gehen und umkehren, sobald Sie in eine Sackgasse geraten (wie bei der Tiefensuche), kann es sein, dass Sie zuerst eine Route finden, die einen gewaltigen Umweg über Seattle macht, bevor sie zurück nach New York führt. Bei einer Breitensuche überprüfen Sie dagegen zuerst alle Stationen, die einen Stopp von Boston entfernt sind. Dann prüfen Sie alle Stationen, die zwei Stopps von Boston entfernt sind. Anschließend prüfen Sie alle Stationen, die drei Stopps von Boston entfernt sind. Das geht so weiter, bis Sie New York finden. Wenn Sie New York auf diese Weise gefunden haben, wissen Sie, dass Sie die Route mit der geringsten Anzahl von Stopps gefunden haben, weil Sie bereits sämtliche Stationen überprüft haben, die weniger Stopps von Boston entfernt liegen, und keine davon New York war.

Warteschlangen

Um BFS zu implementieren, wird eine Datenstruktur benötigt, die als *Warteschlange* (englisch *queue*) bezeichnet wird. Während ein Stapel nach dem LIFO-Prinzip funktioniert, ist es bei der Warteschlange das FIFO-Prinzip (First In, First Out). Eine Warteschlange können Sie sich vorstellen wie die vor öffentlichen Toiletten. Die erste Person, die ansteht, kommt als erste an die Reihe. Eine Warteschlange hat mindestens dieselben `push()`- und `pop()`-Methoden wie ein Stapel. Tatsächlich ist unsere Implementierung für `Queue` (mit einem Python-`deque` als Grundlage) beinahe identisch mit unserer `Stack`-Implementierung, mit den einzigen Änderungen, dass Elemente von der linken Seite des `container`-Objekts statt von der rechten entfernt werden und dass von einer `list` zu einer `deque` gewechselt wird. (Ich verwende hier den Begriff »links«, um den Anfang der zugrundeliegenden Datenstruktur zu bezeichnen.) Die Elemente auf der linken Seite sind die ältesten, die sich noch in der `deque` befinden (was ihre Ankunftszeit angeht), so dass sie als erste entnommen werden.

```
class Queue(Generic[T]):
    def __init__(self) -> None:
        self._container: Deque[T] = Deque()
    @property
    def empty(self) -> bool:
        return not self._container  # not ist für leere Container True
    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.popleft()  # FIFO

    def __repr__(self) -> str:
        return repr(self._container)
```

Listing 2.22 generic_search.py (Fortsetzung)

Tipp

Warum verwendet die Implementierung von `Queue` eine `deque` als zugrundeliegende Datenstruktur, während die Implementierung von `Stack` eine `list` einsetzt? Das hat damit zu tun, wo wir Elemente entnehmen. In einem Stapel fügen wir Elemente auf der rechten Seite hinzu und entnehmen sie dort auch wieder. In einer Warteschlange fügen wir ebenfalls rechts Elemente hinzu, entnehmen sie aber von links. Die Python-Datenstruktur `list` hat eine effiziente `pop()`-Implementierung auf der rechten, aber nicht auf der linken Seite. Bei einer `deque` können Elemente effizient von beiden Seiten entnommen

werden. Entsprechend gibt es bei einer `deque` eine eingebaute Methode namens `popleft()`, aber keine äquivalente Methode bei einer `list`. Es ließen sich sicherlich andere Wege finden, eine `list` als Grundlage für eine Warteschlange zu verwenden, aber sie wären weniger effizient. Ein Element von der linken Seite einer `deque` zu entnehmen, ist eine O(1)-Operation, während es bei einer `list` eine O(*n*)-Operation ist. Im Fall der `list` müsste nach der Entnahme von links jedes Folgeelement nach links verschoben werden, was sie wesentlich weniger effizient macht.

Der BFS-Algorithmus

Faszinierenderweise ist der Algorithmus für eine Breitensuche identisch mit demjenigen für eine Tiefensuche, außer dass `frontier` von einem Stapel in eine Warteschlange umgewandelt wird. Dies ändert wiederum die Reihenfolge, in der Zustände durchsucht werden, und stellt sicher, dass diejenigen, die am nächsten am Startpunkt liegen, zuerst durchsucht werden.

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
 List[T]]) -> Optional[Node[T]]:
    # frontier bezeichnet, wohin wir noch gehen müssen
    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))
    # explored bezeichnet, wo wir schon waren
    explored: Set[T] = {initial}

    # Weitersuchen, solange es noch etwas zu entdecken gibt
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # Wenn wir das Ziel gefunden haben, sind wir fertig
        if goal_test(current_state):
            return current_node
        # Prüfen, wohin wir als Nächstes gehen können und wo wir noch
        # nicht gesucht haben
        for child in successors(current_state):
            if child in explored:  # Bereits durchsuchte Kindknoten überspringen
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # Alles durchsucht, aber nie das Ziel gefunden
```

Listing 2.23 generic_search.py (Fortsetzung)

Wenn Sie versuchen, `bfs()` auszuführen, werden Sie sehen, dass die Methode immer den kürzesten Lösungsweg für das jeweilige Labyrinth findet. Der folgende Versuch wird einfach nach dem vorherigen im Abschnitt `if __name__ == "__main__":` der Datei eingefügt, so dass die Ergebnisse für dasselbe Labyrinth verglichen werden können.

```
# BFS testen
solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test,
 m.successors)
if solution2 is None:
    print("Keine Lösung mit Breitensuche gefunden!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

Listing 2.24 maze.py (Fortsetzung)

Es ist faszinierend, dass Sie einen Algorithmus unverändert lassen und nur die Datenstruktur austauschen können, auf die er zugreift, und radikal unterschiedliche Ergebnisse erhalten. Im Folgenden sehen Sie das Ergebnis des Aufrufs von `bfs()` für dasselbe Labyrinth, für das wir zuvor `dfs()` aufgerufen haben. Beachten Sie, dass der durch die Sternchen markierte Pfad vom Start zum Ziel viel direkter ist als im vorigen Beispiel.

```
S    X X
*X
*      X
*XX      X
* X
* X  X
*X
*
*    X   X
*********G
```

2.2.5 A*-Suche

Es kann sehr zeitaufwendig sein, eine Zwiebel Schicht für Schicht zu schälen, wie es die Breitensuche tut. Genau wie die BFS hat auch eine A*-Suche das Ziel, den kürzesten Pfad von einem Start- zu einem Zielzustand zu finden. Anders als die zuvor beschriebene BFS-Implementierung verwendet eine A*-Suche eine Kombination aus einer Kosten- und

einer Heuristik-Funktion, um die Suche auf Pfade zu konzentrieren, die am wahrscheinlichsten schnell zum Ziel führen.

Die Kostenfunktion g(*n*) bestimmt die Kosten (also den Aufwand), um zu einem bestimmten Zustand zu gelangen. Im Fall unseres Labyrinths würden diese Kosten angeben, wie viele Schritte wir bereits benötigt haben, um zu einem bestimmten Zustand zu gelangen. Die Heuristikfunktion h(*n*) liefert eine Schätzung der Kosten, um vom aktuellen Zustand zum Zielzustand zu gelangen. Für den Fall, dass h(*n*) eine *zulässige Heuristik* ist, kann bewiesen werden, dass der am Ende gefundene Pfad optimal ist. Eine zulässige Heuristik ist eine, die die Kosten zum Erreichen des Ziels niemals überschätzt. Ein Beispiel wäre eine geradlinige Entfernungs-Heuristik auf einer zweidimensionalen Ebene, weil eine gerade Linie immer der kürzeste Pfad ist.[1]

Die Gesamtkosten für jeden in Betracht gezogenen Zustand sind f(*n*), was einfach der Kombination aus g(*n*) und h(*n*) entspricht. Genauer gesagt: f(*n*) = g(*n*) + h(*n*). Bei der Auswahl des nächsten noch unerforschten Pfades wählt eine A*-Suche denjenigen mit dem niedrigsten f(*n*)-Wert. Dadurch unterscheidet sie sich von BFS und DFS.

Prioritätswarteschlangen

Um aus den unerforschten Pfaden den Zustand mit dem niedrigsten f(*n*) auszuwählen, verwendet eine A*-Suche eine *Prioritätswarteschlange* (englisch *priority queue*) als Datenstruktur für die unerforschten Pfade. Eine Prioritätswarteschlange erlegt ihren Elementen eine innere Ordnung auf, so dass das zuerst entnommene Element immer dasjenige mit der höchsten Priorität ist. (In unserem Fall ist das Element mit der höchsten Priorität das mit dem niedrigsten f(*n*).) Normalerweise wird dafür intern ein binärer Heap verwendet, was zu O(lg *n*)-Push- and O(lg *n*)-Pop-Operationen führt.

Pythons Standardbibliothek enthält die Funktionen `heappush()` and `heappop()`, die eine Liste als binären Heap verwalten. Wir können eine Prioritätswarteschlange implementieren, indem wir einen schlanken Wrapper um diese Standardbibliotheksfunktionen erstellen. Unsere Klasse `PriorityQueue` ähnelt unseren Klassen `Stack` und `Queue`, wobei die Methoden `push()` und `pop()` so geändert wurden, dass sie `heappush()` und `heappop()` verwenden.

```
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []
    @property
```

1 Weitere Informationen über Heuristik finden Sie in Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3. Auflage (Pearson, 2010), Seite 94.

```
    def empty(self) -> bool:
        return not self._container  # not ist für leere Container True

    def push(self, item: T) -> None:
        heappush(self._container, item)  # Hinein nach Priorität

    def pop(self) -> T:
        return heappop(self._container)  # Heraus nach Priorität

    def __repr__(self) -> str:
        return repr(self._container)
```

Listing 2.25 generic_search.py (Fortsetzung)

Um die Priorität eines bestimmten Elements gegenüber einem anderen Element seiner Art zu bestimmen, vergleichen `heappush()` und `heappop()` sie mithilfe des Operators `<`. Das ist der Grund, warum wir vorhin `__lt__()` für `Node` implementieren mussten. Ein `Node` wird mit einem anderen verglichen, indem sein jeweiliger f(*n*)-Wert verglichen wird, der einfach die Summe der Eigenschaften `cost` und `heuristic` ist.

Heuristiken

Eine *Heuristik* ist eine intuitive Annahme darüber, wie eine Aufgabe zu lösen ist.[2] Im Fall der Labyrinthlösung versucht eine Heuristik, auszuwählen, welche Position im Labyrinth am besten als nächste durchsucht werden sollte, um schließlich das Ziel zu erreichen. Es handelt sich mit anderen Worten um eine wohlbegründete Vermutung, welche Knoten unter den unerforschten Pfaden dem Ziel am nächsten liegen. Wie bereits erwähnt: Wenn eine für die A*-Suche verwendete Heuristik ein akkurates relatives Ergebnis erzeugt und zulässig ist (also niemals die Entfernung überschätzt), dann liefert A* den kürzesten Pfad. Heuristiken, die kleinere Werte berechnen, führen zum Durchsuchen von mehr Zuständen, während Heuristiken, die dem genauen tatsächlichen Abstand näher kommen (diesen aber nicht überschreiten, was sie unzulässig machen würde), das Durchsuchen von weniger Zuständen zur Folge haben. Deshalb kommen ideale Heuristiken dem wahren Abstand so nahe wie möglich, ohne ihn je zu überschreiten.

Euklidischer Abstand

Wie wir in Geometrie gelernt haben, ist der kürzeste Abstand zwischen zwei Punkten eine gerade Linie. Daher ergibt es Sinn, dass eine geradlinige Heuristik für die Labyrinth-

2 Mehr über Heuristiken für die A*-Pfadsuche finden Sie im Kapitel »Heuristics« in Amit Patels *Amit's Thoughts on Pathfinding*, *http://mng.bz/z7O4*.

Lösungsaufgabe immer zulässig bleibt. Der euklidische Abstand, hergeleitet aus dem Satz des Pythagoras, besagt: `Abstand = √((x-Abstand)² + (y-Abstand)²)`. Bei unseren Labyrinthen entspricht der *x*-Abstand dem Spaltenabstand zwischen zwei Labyrinthpositionen, und der *y*-Abstand entspricht dem Zeilenabstand. Beachten Sie, dass wir dies in *maze.py* implementieren.

```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```

Listing 2.26 maze.py (Fortsetzung)

`euclidean_distance()` ist eine Funktion, die ihrerseits eine Funktion zurückgibt. Sprachen wie Python, die Funktionen erster Klasse unterstützen, bieten dieses interessante Muster. `distance()` führt ein Capturing der Ziel-`MazeLocation` `goal` durch, die an `euclidean_distance()` übergeben wird. Capturing bedeutet, dass die Funktion `distance()` sich jedes Mal, wenn sie aufgerufen wird, (permanent) auf diese Variable beziehen kann. Die zurückgegebene Funktion macht Gebrauch von `goal`, um ihre Berechnungen durchzuführen. Dieses Muster ermöglicht das Schreiben einer Funktion, die weniger Parameter benötigt. Die zurückgegebene Funktion `distance()` nimmt nur die Labyrinth-Startposition als Argument entgegen und »kennt« stets das Ziel.

Abbildung 2.6 veranschaulicht den euklidischen Abstand im Rahmen eines Gitters wie die Straßen von Manhattan.

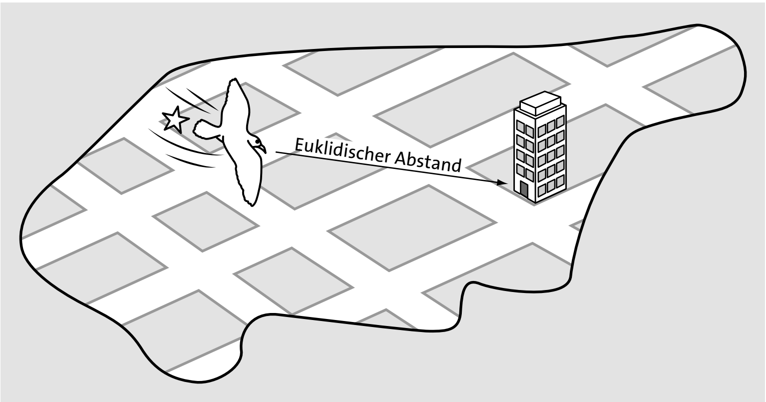


Abbildung 2.6 Der euklidische Abstand ist die Länge einer geraden Linie vom Startpunkt zum Ziel.

Manhattan-Abstand

Der euklidische Abstand ist nicht schlecht, aber für unsere spezifische Aufgabe (ein Labyrinth, in dem Sie sich nur in eine von vier Richtungen bewegen können) geht es sogar noch besser. Der Manhattan-Abstand ist von der Navigation durch die Straßen von Manhattan, dem berühmtesten Stadtteil von New York, abgeleitet, der eine Gitterform aufweist. Um in Manhattan von einem Ort zu irgendeinem anderen zu gelangen, muss man eine bestimmte Anzahl horizontaler Blöcke und eine bestimmte Anzahl vertikaler Blöcke gehen. (In Manhattan gibt es so gut wie keine diagonalen Straßen.) Der Manhattan-Abstand wird einfach berechnet, indem der Zeilenabstand zwischen zwei Labyrinthpositionen zum Spaltenabstand addiert wird. Abbildung 2.7 veranschaulicht den Manhattan-Abstand.

```
def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return (xdist + ydist)
    return distance
```

Listing 2.27 maze.py (Fortsetzung)

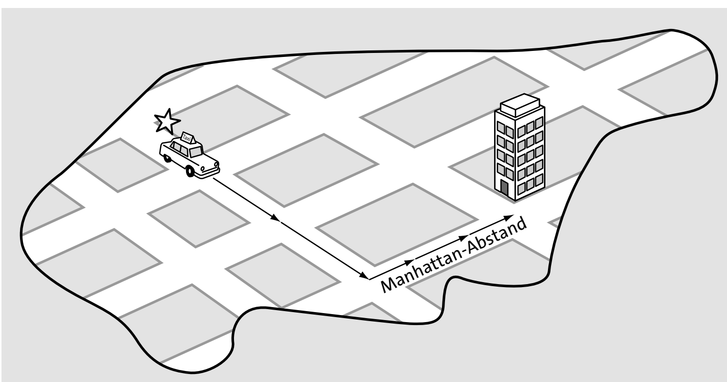


Abbildung 2.7 Beim Manhattan-Abstand gibt es keine Diagonalen. Der Pfad muss entlang waagerechter oder senkrechter Linien verlaufen.

Da diese Heuristik genauer der Realität der Navigation in unseren Labyrinthen entspricht (horizontal und vertikal statt gerader diagonaler Linien), kommt sie dem tatsächlichen Abstand zwischen einer beliebigen Labyrinthposition und dem Ziel näher als der euklidische Abstand. Wenn A* also mit dem Manhattan-Abstand kombiniert wird, ergibt sich für unsere Labyrinthe das Durchsuchen von weniger Zuständen als bei einer A*-Suche, die mit dem euklidischen Abstand kombiniert wird. Die Lösungspfade bleiben

optimal, da der Manhattan-Abstand für Labyrinthe, in denen nur vier Bewegungsrichtungen erlaubt sind, zulässig ist (niemals Entfernungen überschätzt).

Der A*-Algorithmus

Um von BFS zu einer A*-Suche zu gelangen, müssen wir mehrere kleine Änderungen vornehmen. Die erste besteht darin, die unerforschten Pfade in einer Prioritätswarteschlange statt in einer Warteschlange zu speichern. Dadurch werden zuerst Knoten mit dem niedrigsten $f(n)$-Wert entnommen. Die zweite besteht darin, aus der Menge der erforschten Pfade ein Dictionary zu machen. Ein Dictionary ermöglicht es, uns die niedrigsten Kosten ($g(n)$) für jeden Knoten, den wir besuchen können, zu merken. Mit der Heuristikfunktion kann es vorkommen, dass einige Knoten zweimal besucht werden, wenn die Heuristik inkonsistent ist. Wenn der Knoten durch die neue Richtung mit geringeren Kosten zu erreichen ist als beim vorigen Mal, bevorzugen wir die neue Route.

Der Einfachheit halber nimmt die Funktion `astar()` keine Kostenberechnungsfunktion als Parameter entgegen. Stattdessen bewerten wir jeden Schritt in unserem Labyrinth mit dem Kostenfaktor 1. Jedem neuen `Node` werden aufgrund dieser einfachen Formel Kosten zugewiesen, außerdem eine heuristische Punktzahl gemäß einer neuen Funktion namens `heuristic()`, die der Suchfunktion als Parameter übergeben wird. Von diesen Änderungen abgesehen sieht `astar()` der Funktion `bfs()` bemerkenswert ähnlich. Betrachten Sie sie nebeneinander, um sie zu vergleichen.

```
def astar(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
 List[T]], heuristic: Callable[[T], float]) -> Optional[Node[T]]:
    # frontier bezeichnet, wohin wir noch gehen müssen
    frontier: PriorityQueue[Node[T]] = PriorityQueue()
    frontier.push(Node(initial, None, 0.0, heuristic(initial)))
    # explored bezeichnet, wo wir schon waren
    explored: Dict[T, float] = {initial: 0.0}

    # Weitersuchen, solange es noch etwas zu entdecken gibt
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # Wenn wir das Ziel gefunden haben, sind wir fertig
        if goal_test(current_state):
            return current_node
        # Prüfen, wohin wir als Nächstes gehen können und wo wir noch
        # nicht waren
        for child in successors(current_state):
```

```
                new_cost: float = current_node.cost + 1
            # 1 nimmt ein Gitter an; für komplexere Anwendungen ist eine
            # Kostenfunktion erforderlich
                if child not in explored or explored[child] > new_cost:
                    explored[child] = new_cost
                    frontier.push(Node(child, current_node, new_cost,
                     heuristic(child)))
        return None  # Alles durchsucht, aber nie das Ziel gefunden
```

Listing 2.28 generic_search.py

Herzlichen Glückwunsch. Wenn Sie bis hier dabeigeblieben sind, haben Sie nicht nur gelernt, wie man ein Labyrinth löst, sondern Sie haben auch einige generische Suchfunktionen kennengelernt, die Sie in vielen verschiedenen Suchanwendungen einsetzen können. DFS und BFS sind für viele kleinere Datenmengen und Zustandsräume geeignet, wo es nicht so sehr auf Performance ankommt. In manchen Situationen wird DFS schneller sein als BFS, aber BFS hat den Vorteil, immer einen optimalen Pfad zu liefern. Interessanterweise ist die Implementierung von BFS und DFS identisch, außer dass für die unerforschten Pfade eine Warteschlange statt eines Stapels verwendet wird. Die etwas kompliziertere A*-Suche liefert, wenn sie mit einer guten, konsistenten, zulässigen Heuristik kombiniert wird, nicht nur optimale Pfade, sondern ist auch wesentlich schneller als BFS. Und da alle drei Funktionen generisch implementiert wurden, liegt ihre Verwendung nur ein `import generic_search` weit entfernt.

Probieren Sie `astar()` im Labyrinth-Test-Abschnitt von von *maze.py* mit demselben Labyrinth aus.

```
# A* testen
distance: Callable[[MazeLocation], float] = manhattan_distance(m.goal)
solution3: Optional[Node[MazeLocation]] = astar(m.start, m.goal_test,
 m.successors, distance)
if solution3 is None:
    print("Keine Lösung mit A* gefunden!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print(m)
```

Listing 2.29 maze.py (Fortsetzung)

Die Ausgabe unterscheidet sich interessanterweise etwas von `bfs()`, obwohl `bfs()` und `astar()` optimale Pfade (mit identischer Länge) finden. Aufgrund seiner Heuristik strebt

die Funktion `astar()` sofort eine Diagonale in Richtung Ziel an. Sie wird schließlich weniger Zustände durchsuchen als `bfs()`, was zu einer besseren Performance führt. Fügen Sie einen Zustandszähler hinzu, wenn Sie das selbst überprüfen möchten.

```
S**  X X
 X**
   *   X
 XX*     X
  X*
  X**X
 X  ****
       *
     X * X
       **G
```

2.3 Missionare und Kannibalen

Drei Missionare und drei Kannibalen befinden sich am Westufer eines Flusses. Sie haben ein Kanu, das zwei Personen aufnehmen kann, und alle müssen den Fluss überqueren, um zum Ostufer des Flusses zu gelangen. Es darf nie mehr Kannibalen als Missionare am selben Ufer des Flusses geben, sonst essen die Kannibalen die Missionare. Zudem muss das Kanu mindestens eine Person an Bord haben, um den Fluss zu überqueren. Welche Überquerungsreihenfolge bringt alle erfolgreich ans andere Ufer? Abbildung 2.8 veranschaulicht die Aufgabe.

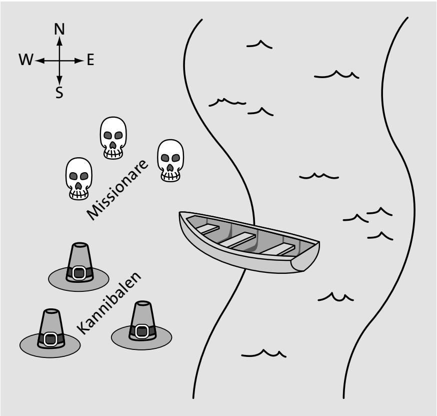


Abbildung 2.8 Die Missionare und die Kannibalen müssen ihr einziges Kanu benutzen, um alle von Westen nach Osten über den Fluss zu bringen. Wenn jemals die Kannibalen in der Überzahl sind, essen sie die Missionare.

2.3.1 Die Aufgabe darstellen

Wir stellen die Aufgabe dar, indem wir eine Struktur speichern, die sich das Geschehen am Westufer merkt. Wie viele Missionare und wie viele Kannibalen befinden sich am Westufer? Befindet sich das Boot am Westufer? Mit diesem Wissen können wir herausfinden, was am Ostufer los ist, denn alles, was sich nicht am Westufer befindet, ist dort.

Zuerst erstellen wir eine kleine Hilfsvariable, um uns die maximale Anzahl von Missionaren oder Kannibalen zu merken. Dann definieren wir die Hauptklasse.

```
from __future__ import annotations
from typing import List, Optional
from generic_search import bfs, Node, node_to_path

MAX_NUM: int = 3

class MCState:
    def __init__(self, missionaries: int, cannibals: int, boat: bool) -> None:
        self.wm: int = missionaries # Missionare am Westufer
        self.wc: int = cannibals # Kannibalen am Westufer
        self.em: int = MAX_NUM - self.wm  # Missionare am Ostufer
        self.ec: int = MAX_NUM - self.wc  # Kannibalen am Ostufer
        self.boat: bool = boat  # Boot am Westufer oder nicht?

    def __str__(self) -> str:
        return ("Am Westufer sind {} Missionare und {} Kannibalen.\n"
                "Am Ostufer sind {} Missionare und {} Kannibalen.\n"
                "Das Boot ist am {}ufer.")\
            .format(self.wm, self.wc, self.em, self.ec, ("West" if self.boat
             else "Ost"))
```

Listing 2.30 missionaries.py

Die Klasse `MCState` wird mit der Anzahl der Missionare und Kannibalen am Westufer sowie mit der Position des Boots initialisiert. Sie kann auch für ihre eigene lesbare Ausgabe sorgen, was später bei der Ausgabe der Lösung dieser Ausgabe hilfreich sein wird.

Aufgrund der Vorgaben unserer bestehenden Suchfunktionen müssen wir eine Funktion definieren, die testet, ob ein Zustand der Zielzustand ist, sowie eine Funktion, die die Nachfolger jedes Zustands findet. Die Zieltestfunktion ist wie beim Labyrinth-Lösungs-Problem ziemlich einfach. Das Ziel ist erreicht, wenn wir einen erlaubten Zustand erreichen, bei dem sich alle Missionare und Kannibalen am Ostufer befinden. Wir fügen dies als Methode zu `MCState` hinzu.


```
def goal_test(self) -> bool:
    return self.is_legal and self.em == MAX_NUM and self.ec == MAX_NUM
```

Listing 2.31 missionaries.py (Fortsetzung)

Um eine Nachfolgerfunktion zu schreiben, ist es nötig, alle möglichen Züge durchzugehen, die von einem Ufer zum anderen führen, und dann zu prüfen, ob diese Zustände zu einem erlaubten Zustand führen. Denken Sie daran, dass ein erlaubter Zustand einer ist, in dem nicht mehr Kannibalen als Missionare an einem der Ufer sind. Um dies zu ermitteln, können wir eine Hilfseigenschaft (als Methode von `MCState`) definieren, die überprüft, ob ein Zustand erlaubt ist.

```
@property
def is_legal(self) -> bool:
    if self.wm < self.wc and self.wm > 0:
        return False
    if self.em < self.ec and self.em > 0:
        return False
    return True
```

Listing 2.32 missionaries.py (Fortsetzung)

Die eigentliche Nachfolgerfunktion ist etwas ausführlicher, damit ihre Funktionsweise klarer wird. Sie sammelt jede mögliche Kombination von einer oder zwei Personen, die den Fluss von dem Ufer aus überqueren, an dem sich das Kanu gerade befindet. Nachdem sie alle möglichen Züge hinzugefügt hat, filtert sie mithilfe einer List Comprehension diejenigen heraus, die tatsächlich erlaubt sind. Auch dies ist eine Methode von `MCState`.

```
def successors(self) -> List[MCState]:
    sucs: List[MCState] = []
    if self.boat: # Boot am Westufer
        if self.wm > 1:
            sucs.append(MCState(self.wm - 2, self.wc, not self.boat))
        if self.wm > 0:
            sucs.append(MCState(self.wm - 1, self.wc, not self.boat))
        if self.wc > 1:
            sucs.append(MCState(self.wm, self.wc - 2, not self.boat))
        if self.wc > 0:
            sucs.append(MCState(self.wm, self.wc - 1, not self.boat))
```

```
        if (self.wc > 0) and (self.wm > 0):
            sucs.append(MCState(self.wm - 1, self.wc - 1, not self.boat))
    else: # Boot am Ostufer
        if self.em > 1:
            sucs.append(MCState(self.wm + 2, self.wc, not self.boat))
        if self.em > 0:
            sucs.append(MCState(self.wm + 1, self.wc, not self.boat))
        if self.ec > 1:
            sucs.append(MCState(self.wm, self.wc + 2, not self.boat))
        if self.ec > 0:
            sucs.append(MCState(self.wm, self.wc + 1, not self.boat))
        if (self.ec > 0) and (self.em > 0):
            sucs.append(MCState(self.wm + 1, self.wc + 1, not self.boat))
    return [x for x in sucs if x.is_legal]
```

2.3.2 Lösung

Wir haben nun alle Zutaten beisammen, um die Aufgabe zu lösen. Wenn wir eine Aufgabe mithilfe der Suchfunktionen `bfs()`, `dfs()` und `astar()` lösen, erhalten wir, wie Sie sicher noch wissen, einen `Node` zurück, den wir schließlich mit `node_to_path()` in eine Liste von Zuständen umwandeln, die zu einer Lösung führt. Was wir noch brauchen, ist eine Möglichkeit, diese Liste in eine verständlich dargestellte Reihe von Schritten umzuwandeln, die die Aufgabe mit den Missionaren und Kannibalen lösen.

Die Funktion `display_solution()` konvertiert einen Lösungspfad in eine Textausgabe – eine für Menschen lesbare Lösung der Aufgabe. Sie arbeitet, indem sie über alle Zustände des Lösungspfads iteriert und sich dabei auch den vorigen Zustand merkt. Sie betrachtet den Unterschied zwischen dem vorigen Zustand und demjenigen, der gerade in der Iteration an der Reihe ist, um herauszufinden, wie viele Missionare und Kannibalen den Fluss in welche Richtung überquert haben.

```
def display_solution(path: List[MCState]):
    if len(path) == 0: # Gültigkeitsprüfung
        return
    old_state: MCState = path[0]
    print(old_state)
    for current_state in path[1:]:
        if current_state.boat:
            print("{} Missionare und {} Kannibalen vom Ostufer zum Westufer
             transportiert.\n"
                 .format(old_state.em - current_state.em, old_state.ec -
                  current_state.ec))
        else:
            print("{} Missionare und {} Kannibalen vom Westufer zum Ostufer
             transportiert.\n"
                 .format(old_state.wm - current_state.wm, old_state.wc -
                  current_state.wc))
        print(current_state)
        old_state = current_state
```

Listing 2.33 missionaries.py (Fortsetzung)

Die Funktion `display_solution()` macht sich die Tatsache zunutze, dass die Klasse `MCState` mittels `__str__()` eine übersichtlich formatierte Zusammenfassung ihrer selbst liefern kann.

Was wir zuletzt noch tun müssen, ist, die Aufgabe mit den Missionaren und Kannibalen tatsächlich zu lösen. Um dies zu tun, können wir bequemerweise eine der Suchfunktionen wiederverwenden, die wir bereits implementiert haben, da wir sie generisch implementiert haben. Diese Lösung verwendet `bfs()` (weil die Verwendung von `dfs()` es erforderlich machen würde, referentiell unterschiedliche Zustände als gleich zu markieren, und `astar()` eine Heuristik benötigen würde).

```
if __name__ == "__main__":
    start: MCState = MCState(MAX_NUM, MAX_NUM, True)
    solution: Optional[Node[MCState]] = bfs(start, MCState.goal_test,
     MCState.successors)
    if solution is None:
        print("Keine Lösung gefunden!")
    else:
        path: List[MCState] = node_to_path(solution)
        display_solution(path)
```

Listing 2.34 missionaries.py (Fortsetzung)

Es ist großartig, zu sehen, wie flexibel unsere generischen Suchfunktionen sein können. Sie können einfach angepasst werden, um viele unterschiedliche Aufgaben zu lösen. Sie sollten eine Ausgabe wie die folgende (gekürzte) sehen:

```
Am Westufer sind 3 Missionare und 3 Kannibalen.
Am Westufer sind 0 Missionare und 0 Kannibalen.
Das Boot ist am Westufer.
```

```
0 Missionare und 2 Kannibalen vom Westufer zum Ostufer gebracht.
Am Westufer sind 3 Missionare und 1 Kannibalen.
Am Ostufer sind 0 Missionare und 2 Kannibalen.
Das Boot ist am Ostufer.
0 Missionare und 1 Kannibalen vom Ostufer zum Westufer gebracht.
…
Am Westufer sind 0 Missionare und 0 Kannibalen.
Am Ostufer sind 3 Missionare und 3 Kannibalen.
Das Boot ist am Ostufer.
```

2.4 Anwendungen im Alltag

In jeder nützlichen Software spielt Suche eine Rolle. In einigen Fällen ist sie das zentrale Element (Google-Suche, Spotlight, Lucene); bei anderen ist sie die Grundlage der zugrundeliegenden Datenspeicherung. Den korrekten Suchalgorithmus für eine Datenstruktur zu kennen, ist unerlässlich für die Performance. Zum Beispiel wäre es sehr kostspielig, die lineare Suche statt der binären Suche auf eine sortierte Datenstruktur anzuwenden.

A* ist einer der am weitesten verbreiteten Pfad-Such-Algorithmen. Er wird nur von Algorithmen überboten, die Vorausberechnungen im Suchraum durchführen. Für eine blinde Suche wurde A* noch in keinem Szenario nachweislich besiegt, und das hat den Algorithmus zu einer wichtigen Komponente aller erdenklichen Aufgaben gemacht, von der Routenplanung über das Finden des kürzesten Weges bis hin zum Parsen einer Programmiersprache. Fast jede Kartensoftware, die Routenplanung bietet (denken Sie an Google Maps), verwendet den Dijkstra-Algorithmus (von dem A* eine Variante ist) zur Navigation. (Mehr über den Dijkstra-Algorithmus erfahren Sie in Kapitel 4, »Graphenprobleme«.) Wenn ein KI-Charakter in einem Spiel ohne menschlichen Eingriff den kürzesten Pfad von einem Ende der Spielwelt zum anderen findet, verwendet er wahrscheinlich A*.

Breitensuche und Tiefensuche bilden oft die Grundlage komplexerer Suchalgorithmen wie uniforme Kostensuche und Backtracking-Suche (die Sie im nächsten Kapitel sehen werden). Breitensuche ist oft ein ausreichendes Verfahren, um den kürzesten Weg in einem relativ kleinen Graphen zu finden. Aber aufgrund der Ähnlichkeit zu A* ist es einfach, den Algorithmus durch A* zu ersetzen, wenn eine gute Heuristik für einen größeren Graphen existiert.

2.5 Übungsaufgaben

1. Zeigen Sie den Performancevorteil der binären Suche gegenüber der linearen Suche, indem Sie eine Liste von einer Million Zahlen erzeugen und die Zeit stoppen, wie lange die in diesem Kapitel definierten Funktionen `linear_contains()` und `binary_contains()` brauchen, um verschiedene Zahlen in der Liste zu finden.
2. Fügen Sie zu `dfs()`, `bfs()` und `astar()` einen Zähler hinzu, um herauszufinden, wie viele Zustände sie jeweils für dasselbe Labyrinth durchsuchen. Vergleichen Sie die Anzahlen für 100 verschiedene Labyrinthe, um statistisch relevante Ergebnisse zu erhalten.
3. Finden Sie eine Lösung der Aufgabe mit den Missionaren und Kannibalen für eine ungleiche Anzahl von Missionaren und Kannibalen. Tipp: Sie müssen wahrscheinlich die Methoden `__eq__()` und `__hash__()` von `MCState` überschreiben.

Auf einen Blick

Inhalt

7 Einfache neuronale Netzwerke 185

8 Adversarial Search 219

9 Sonstige Aufgaben 245

Anhang 263