2 Search problems

livebook.manning.com

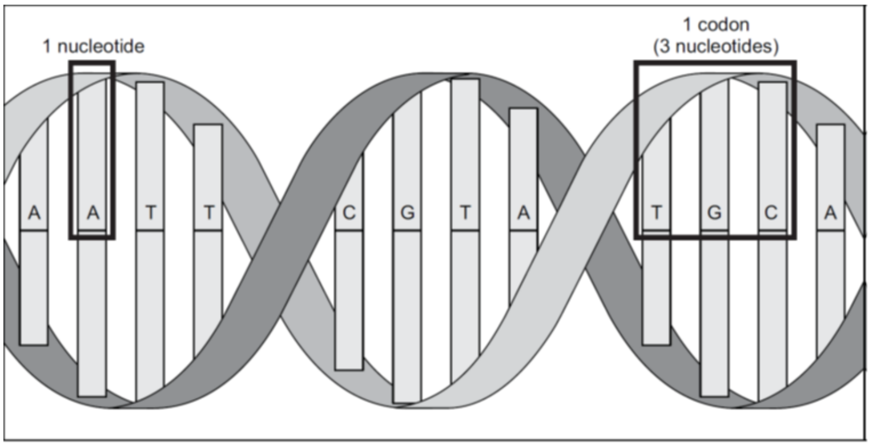
99

"Search" is such a broad term that this entire book could be called "Classic Search Problems in Python." This chapter is about core search algorithms that every programmer should know. It does not claim to be comprehensive, despite the declaratory title.

2.1 DNA search

Genes are commonly represented in computer software as a sequence of the characters A, C, G, and T. Each letter represents a *nucleotide*, and the combination of three nucleotides is called a *codon*. This is illustrated in figure 2.1. A codon codes for a specific amino acid that together with other amino acids can form a *protein*. A classic task in bioinformatics software is to find a particular codon within a gene.

Figure 2.1 A nucleotide is represented by one of the letters A, C, G, and T. A codon is composed of three nucleotides, and a gene is composed of multiple codons.



Part of a gene

2.1.1 Storing DNA

We can represent a nucleotide as a simple `IntEnum` with four cases.

Listing 2.1 dna_search.py

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

copy

`Nucleotide` is of type `IntEnum` instead of just `Enum`, because `IntEnum` gives us comparison operators (`<`, `>=`, etc.) "for free." Having these operators in a data type is required for the search algorithms we are going to implement to be able to operate on it. `Tuple` and `List` are imported from the `typing` package to assist with type hints.

Codons can be defined as a tuple of three `Nucleotide` s. And finally, a gene may be defined as a list of `Codon` s.

Listing 2.2 dna_search.py continued

```
Codon = Tuple[Nucleotide, Nucleotide, Nucleotide]  # type alias for codons

Gene = List[Codon]  # type alias for genes
```

copy

Note

Although we will later need to compare one `Codon` to another, we do not need to define a custom class with the < operator explicitly implemented for `Codon`. This is because Python has built-in support for comparisons between tuples that are composed of types that are also comparable.

Typically, genes that you find on the internet will be in a file format that contains a giant string representing all of the nucleotides in the gene's sequence. We will define such a string for an imaginary gene and call it `gene_str`.

Listing 2.3 dna_search.py continued

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

copy

We will also need a utility function to convert a `str` into a `Gene`.

Listing 2.4 dna_search.py continued

```
def string_to_gene(s: str) -> Gene:
    gene: Gene = []
    for i in range(0, len(s), 3):
        if (i + 2) >= len(s):

            return gene


        codon: Codon = (Nucleotide[s[i]], Nucleotide[s[i + 1]], Nucleotide[s[i + 2]])
        gene.append(codon)

    return gene
```

copy

`string_to_gene()` continually goes through the provided `str` and converts its next three characters into `Codon` s that it adds to the end of a new `Gene`. If it finds that there is no `Nucleotide` two places into the future of the current place in `s` that it is examining (see the `if` statement within the loop), then it knows it has reached the end of an incomplete gene, and it skips over those last one or two nucleotides.

`string_to_gene()` can be used to convert the `str` `gene_str` into a `Gene`.

Listing 2.5 dna_search.py continued

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

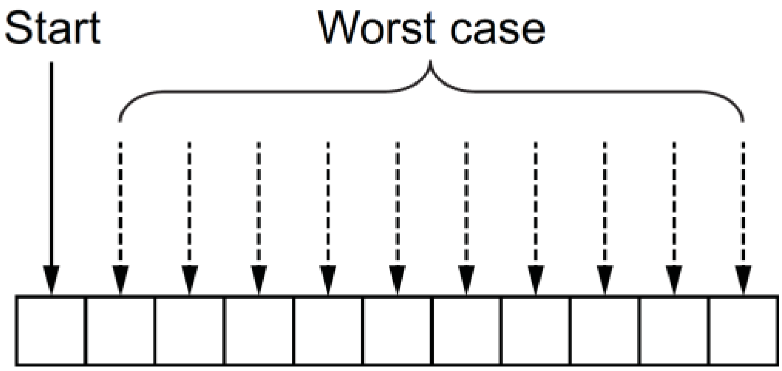
copy

2.1.2 Linear search

One basic operation we may want to perform on a gene is to search it for a particular codon. The goal is to simply find out whether the codon exists within the gene or not.

A linear search goes through every element in a search space, in the order of the original data structure, until what is sought is found or the end of the data structure is reached. In effect, a linear search is the most simple, natural, and obvious way to search for something. In the worst case, a linear search will require going through every element in a data structure, so it is of O(n) complexity, where n is the number of elements in the structure. This is illustrated in figure 2.2.

Figure 2.2 In the worst case of a linear search, you'll sequentially look through every element of the array.



It is trivial to define a function that performs a linear search. It simply must go through every element in a data structure and check for its equivalence to the item being sought. The following code defines such a function for a `Gene` and a `Codon` and then tries it out for `my_gene` and `Codon` s called `acg` and `gat` .

Listing 2.6 dna_search.py continued

```
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))
print(linear_contains(my_gene, gat))
```

copy

Note

This function is for illustrative purposes only. The Python built-in sequence types (`list` , `tuple` , `range`) all implement the `__contains__()` method which allows one to do a search for a specific item in them simply using the `in` operator. In fact, the `in` operator can be used with any type that implements `__contains__()` . So, for instance one could search `my_gene` for `acg` and print out the result by writing `print(acg in my_gene)` .

2.1.3 Binary search

There is a faster way to search than looking at every element, but it requires us to know something about the order of the data structure ahead of time. If we know that the structure is sorted, and we can instantly access any item within it by its index, then we can perform a binary search. Based on this criteria, a sorted Python `list` is a perfect candidate for a binary search.

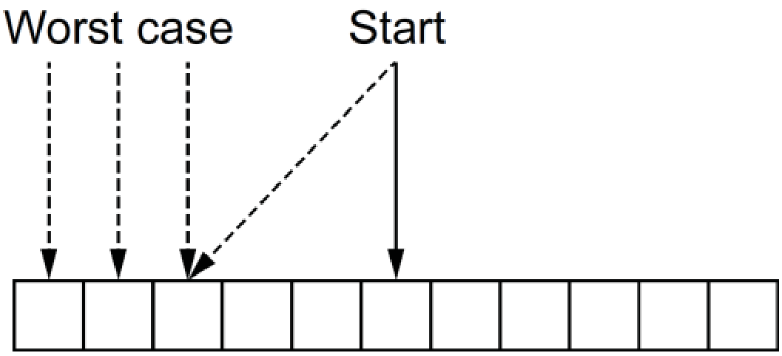
A binary search works by looking at the middle element in a sorted range of elements, comparing it to the element sought, and then reducing the range by half based on that comparison, and starting the process over again. Let's look at a concrete example.

Suppose we have a `list` of alphabetically sorted words like `["cat", "dog", "kangaroo", "llama", "rabbit", "rat", "zebra"]` and we are searching for the word "rat":

1. We could determine that the middle element in this seven-word list is "llama."
2. We could determine that "rat" comes after "llama" alphabetically, so it must be in the approximately half of the list that comes after "llama." (If we had found "rat" in this step, we could have returned its location, or if we had found that our word came before the middle word we were checking, we could be assured that it was in the approximately half of the list before "llama.")
3. We could rerun steps 1 and 2 for the half of the list that we know "rat" is still possibly in. In effect, this half becomes our new base list. Steps 1 through 3 continually run until "rat" is found or the range we are looking in no longer contains any elements to search, meaning "rat" does not exist within the word list.

Figure 2.3 illustrates a binary search. Notice that it does not involve searching every element, unlike a linear search.

Figure 2.3 In the worst case of a binary search, you'll look through just lg(n) elements of the list.



A binary search continually reduces the search space by half, so it has a worst-case runtime of O(lg n). There is a sort-of catch, though. Unlike a linear search, a binary search requires a sorted data structure to search through. Sorting takes time. In fact, sorting takes O(n lg n) time for the best sorting algorithms. If we are only going to run our search once, and our original data structure is unsorted, it probably makes sense to just do a linear search. However, if the search is going to be performed many times, the time cost of doing the sort itself is worth it to reap the benefit of the greatly reduced time cost of each individual search.

Writing a binary search function for a gene and a codon is not unlike writing one for any other type of data, because the `Codon` type can be compared to others of its type, and the `Gene` type is just a `list`.

Listing 2.7 dna_search.py continued

```
def binary_contains(gene: Gene, key_codon: Codon) -> bool:
    low: int = 0

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

copy

Let’s walk through this function line by line.

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

copy

We start by looking at a range that encompasses the entire list (gene).

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

copy

We keep searching as long as there is a still a range to search within. When `low` is greater than `high`, it means that there are no longer any slots to look at within the list.

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

copy

We calculate the middle, `mid`, by using integer division and the simple mean formula you learned in grade school.

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

copy

If the element we are looking for is after the middle element of the range we are looking at, then we modify the range that we will look at during the next iteration of the loop by moving `low` to be one past the current middle element. This is where we halve the range for the next iteration.

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

copy

Similarly, we halve in the other direction when the element we are looking for is less than the middle element.

```
else:
    return True
```

copy

If the element in question is not less than or greater than the middle element, that means we found it! And, of course, if the loop ran out of iterations, we return `False` (not reproduced here), indicating that it was never found.

We can try running our function with the same gene and codon, but we must remember to sort first:

Listing 2.8 dna_search.py continued

```
my_sorted_gene: Gene = sorted(my_gene)
print(binary_contains(my_sorted_gene, acg))

print(binary_contains(my_sorted_gene, gat))
```

copy

TIP

You can build a performant binary search using the Python standard library's `bisect` module: https://docs.python.org/3/library/bisect.html

2.1.4 A generic example

IMPORTANT

Before proceeding with the book you will need to install the `typing_extensions` module via either `pip install typing_extensions` or `pip3 install typing_extensions` depending on how your Python interpreter is configured. We need this module for the `Protocol` type, which will be in the standard library in a future version of Python (as specified by PEP 544). Therefore, in a future version of Python, importing the `typing_extensions` module should be unnecessary and one will be able to `from typing import Protocol` instead of `from typing_extensions import Protocol` .

The functions `linear_contains()` and `binary_contains()` can be generalized to work with almost any Python sequence. These generalized versions are nearly identical to the versions you saw before, with only some names and type hints changed.

Note

There are many imported types in the following code listing, because we will be reusing the file `generic_search.py` for many further generic search algorithms in this chapter and we wanted to get the imports out of the way.

Listing 2.9 generic_search.py

```
from __future__ import annotations
from typing import TypeVar, Iterable, Sequence, Generic, List, Callable, Set, Deque, Dict, Any, Optional
from typing_extensions import Protocol
from functools import total_ordering
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:  # while there is still a search space
        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
```

copy

Now you can try doing searches on other types of data. These functions can be reused for almost any Python collection. That is the power of writing one's code generically. The only unfortunate element of this example is the convoluted hoops that had to be jumped through for Python's type hints in the form of the `Comparable` class. A `Comparable` type is a type that implements the comparison operators (<, >=, etc.). There should be a more succinct way in future versions of Python to create a type hint for types that implement these common operators.

2.2 Maze solving

176

Finding a path through a maze is analogous to many common search problems in computer science. Why not literally find a path through a maze then, to illustrate the breadth-first search, depth-first search, and A* algorithms?

Our maze will be a two-dimensional grid of `Cell` s. A `Cell` is an enum with `str` values where “ “ will represent an empty space and ” `X`” will represent a blocked space. There are also various other cases for illustrative purposes when printing a maze.

Listing 2.10 maze.py

```
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 = "*"
```

copy

Once again, we are getting a large number of imports out of the way. Note that the last import (from `generic_search`) is of symbols we have not yet defined. It is included here for convenience, but you may want to comment it out until you are ready.

We'll need a way to refer to an individual location in the maze. This will simply be a `NamedTuple` with properties representing the row and column of the location in question.

Listing 2.11 maze.py continued

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

    column: int
```

copy

2.2.1 Generating a random maze

Our `Maze` class will internally keep track of a grid (a list of lists) representing its state. It will also have instance variables for the number of rows, number of columns, start location, and goal location. Its grid will be randomly filled with blocked cells.

The maze that is generated should be fairly sparse so that there is almost always a path from a given starting location to a given goal location (this is for testing our algorithms, after all). We'll let the caller of a new maze decide on the exact sparseness, but we will provide a default value of 20% blocked. When a random number beats the threshold of the `sparseness` parameter in question, we will simply replace an empty space with a wall. If we do this for every possible place in the maze, statistically the sparseness of the maze as a whole will approximate the `sparseness` parameter supplied.

Listing 2.12 maze.py continued

```
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:
        # initialize basic instance variables

        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # fill the grid with empty cells

        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)] for r in range(rows)]
        # populate the grid with blocked cells

        self._randomly_fill(rows, columns, sparseness)
        # fill the start and goal locations in

        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
```

copy

Now that we have a maze, we also want a way to print it succinctly to the console. We want its characters to be close together so it looks like a real maze.

Listing 2.13 maze.py continued

```
def __str__(self) -> str:
    output: str = ""

    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"

    return output
```

copy

Go ahead and test these maze functions.

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

copy

2.2.2 Miscellaneous maze minutiae

It will be handy later to have a function that checks whether we have reached our goal during the search. In other words, we want to check whether a particular `MazeLocation` that the search has reached is the goal. We add a method to `Maze`.

Listing 2.14 maze.py continued

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

copy

How can one move within our mazes? Let’s say that one can move horizontally and vertically one space at a time from a given space in the maze. Using these criteria, a `successors()` function can find the possible next locations from a given `MazeLocation`. However, the `successors()` function will differ for every `Maze` because every `Maze` has a different size and set of walls. Therefore, we will define it as a method on `Maze`.

Listing 2.15 maze.py continued

```
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
```

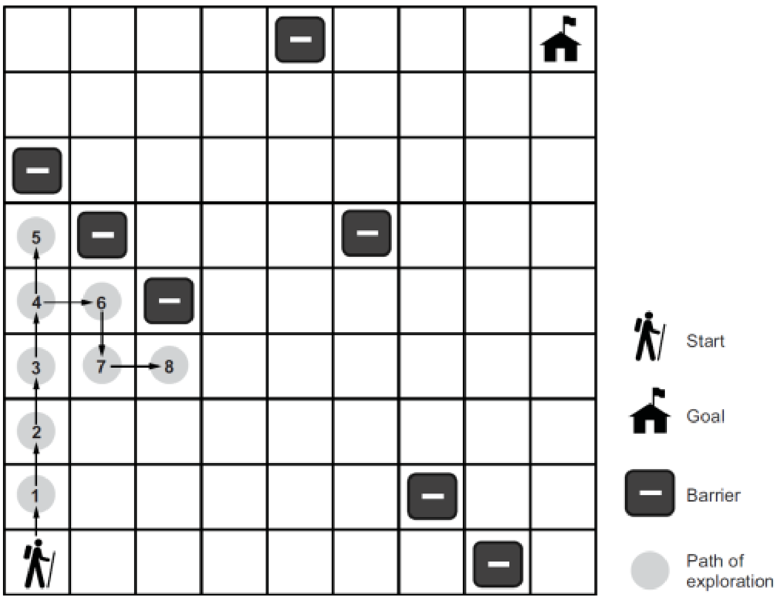
copy

`successors()` simply checks above, below, to the right, and to the left of a `MazeLocation` in a `Maze` to see if it can find empty spaces that can be gone to from that location. It also avoids checking locations beyond the edges of the `Maze`. Every possible `MazeLocation` that it finds it puts into a list that it ultimately returns to the caller.

2.2.3 Depth-first search

A depth-first search (DFS) is what its name suggests—a search that goes as deeply as it can before backtracking to its last decision point if it reaches a dead end. We will implement a generic depth-first search that can solve our maze problem. It will also be reusable for other problems. Figure 2.4 illustrates an in-progress depth-first search of a maze.

Figure 2.4 In depth-first search, the search proceeds along a continuously deeper path until it hits a barrier and must backtrack to the last decision point.



Stacks

The depth-first search algorithm relies on a data structure known as a *stack*. (If you read about stacks in chapter 1, feel free to skip this section). A stack is a data structure that operates under the Last-In-First-Out (LIFO) principle. Imagine a stack of papers. The last paper placed on top of the stack is the first paper pulled off the stack. It is common for a stack to be implemented on top of a more primitive data structure like a list. We will implement our stack on top of Python's `list` type.

Stacks generally have at least two operations:

- `push()` —Places an item on top of the stack
- `pop()` —Removes the item on the top of the stack and returns it

We will implement both of these, as well as an `empty` property to check if the stack has any more items in it. We will add the code for the stack back in our `generic_search.py` file where we already have completed several necessary imports.

Listing 2.16 generic_search.py continued

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

    def empty(self) -> bool:
        return not self._container

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

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

copy

Note that implementing a stack using a Python `list` is as simple as always appending items onto its right end, and always removing items from its extreme right end. The `pop()` method on `list` will fail if there are no longer any items in the list, so `pop()` will fail on a `Stack` if it is empty as well.

The DFS algorithm

We will need one more little tidbit before we can get to implementing DFS. We need a `Node` class that will be used to keep track of how we got from one state to another state (or from one place to another place) as we search. You can think of a `Node` as a wrapper around a state. In the case of our maze-solving problem, those states are of type `MazeLocation` . We'll call the `Node` that a state came from its `parent` . We will also define our `Node` class as having `cost` and `heuristic` properties and with `__lt__()` implemented, so we can reuse it later in the A* algorithm.

Listing 2.17 generic_search.py continued

```
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)
```

copy

TIP

The `Optional` type indicates that a value of a parameterized type may be referenced by the variable, or the variable may reference `None` .

TIP

At the top of the file, the `from __future__ import annotations` allows `Node` to reference itself in the type hints of its methods. Without it, one must put the type hint in quotes as a string (e.g. `‘Node’`). In future versions of Python, importing `annotations` will be unnecessary. See PEP 563 “Postponed Evaluation of Annotations” for more information: https://www.python.org/dev/peps/pep-0563/

An in-progress depth-first search needs to keep track of two data structures: the stack of states (or “places”) that we are considering searching, which we will call the `frontier` ; and the set of states that we have already searched, which we will call `explored` . As long as there are more states to visit in the frontier, DFS will keep checking whether they are the goal (if a state is the goal, it will stop and return it) and adding their successors to the frontier. It will also mark each state that has already been searched as explored, so that it does not get caught in a circle, reaching states that have prior visited states as successors. If the frontier is empty, it means there is nowhere left to search.

Listing 2.18 generic_search.py continued

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]]) -> Optional[Node[T]]:


    frontier: Stack[Node[T]] = Stack()
    frontier.push(Node(initial, None))


    explored: Set[T] = {initial}

    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state


        if goal_test(current_state):
            return current_node


        for child in successors(current_state):
            if child in explored:

                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None
```

copy

If `dfs()` is successful, it returns the `Node` encapsulating the goal state. The path from the start to the goal can be reconstructed by working backward from this `Node` and its priors using the `parent` property.

Listing 2.19 generic_search.py continued

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]


    while node.parent is not None:
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

copy

For display purposes, it will be useful to mark up the maze with the successful path, the start state, and the goal state. It will also be useful to be able to remove a path, so that we can try different search algorithms on the same maze. The following two methods should be added to the `Maze` class in `maze.py` .

Listing 2.20 maze.py continued

```
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
```

copy

It has been a long journey, but we are finally ready to solve the maze.

Listing 2.21 maze.py continued

```
if __name__ == "__main__":

    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_test, m.successors)
    if solution1 is None:
        print("No solution found using depth-first search!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print(m)
        m.clear(path1)
```

copy

A successful solution will look something like this:

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

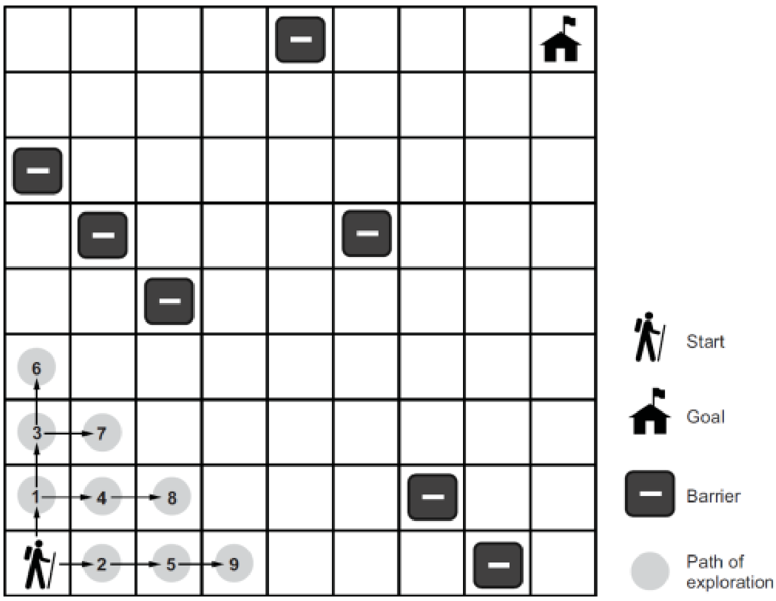
copy

The asterisks represent the path that our depth-first search function found from the start to the goal. Remember, because each maze is randomly generated, not every maze has a solution.

2.2.4 Breadth-first search

You may notice that the solution paths to the mazes found by depth-first traversal seem unnatural. They are usually not the shortest paths. Breadth-first search (BFS) always finds the shortest path by systematically looking one layer of nodes further away from the start state each iteration of the search. There are particular problems in which a depth-first search is likely to find a solution prior to a breadth-first search, and vice versa. Therefore, choosing between the two is sometimes a trade-off between the possibility of finding a solution quickly and the certainty of finding the shortest path to the goal (if one exists). Figure 2.5 illustrates an in-progress breadth-first search of a maze.

Figure 2.5 In a breadth-first search, the closest elements to the starting location are searched first.



To understand why a depth-first search sometimes returns a result faster than a breadth-first search, imagine looking for a marking on a particular layer of an onion. A searcher using a depth-first strategy may plunge a knife into the center of the onion and haphazardly examine the chunks cut out. If the marked layer happens to be near the chunk cut out, there is a chance that the searcher will find it more quickly than another searcher using a breadth-first strategy who painstakingly peels back the onion one layer at a time.

To get a better picture of why breadth-first search always finds the shortest solution path where one exists, consider trying to find the path with the fewest number of stops between Boston and New York by train. If you keep going in the same direction and backtracking when you hit a dead end (as in depth-first search), you may first find a route all the way to Seattle before it connects back to New York. However, in a breadth-first search, you will first check all of the stations one stop away from Boston. Then you will check all of the stations two stops away from Boston. Then you will check all of the stations three stops away from Boston. This will keep going until you find New York. Therefore, when you do find New York, you will know you have found the route with the fewest stops, because you already checked all of the stations that are fewer stops away from Boston, and none of them were New York.

Queues

To implement BFS, a data structure known as a *queue* is required. Whereas a stack is LIFO, a queue is FIFO—First-In-First-Out. A queue is like a line to use a restroom. The first person who got in line goes to the restroom first. At a minimum, a queue has the same `push()` and `pop()` methods as a stack. In fact, our implementation for `Queue` (backed by a Python `deque`) is almost identical to our implementation of `Stack` , with the only change being the removal of elements from the left end of the `_container` instead of the right end and the switch from a `list` to a `deque` (we use the word “left” here to mean the beginning of the backing store). The elements on the left end are the oldest elements still in the `deque` (in terms of arrival time), so they are the first elements popped.

Listing 2.21 generic_search.py continued

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

    def empty(self) -> bool:
        return not self._container

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

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

copy

TIP

Why did the implementation of `Queue` use a `deque` as its backing store, while the implementation of `Stack` used a `list` as its backing store? It has to do with where we pop. In a stack, we push to the right and pop from the right. In a queue we push to the right as well, but we pop from the left. The Python `list` data structure has efficient pops from the right, but not from the left. A `deque` can efficiently pop from either side. As a result, there is a built-in method on `deque` called `popleft(`) but no equivalent method on `list` . One can certainly find other ways to use a `list` as the backing store for a queue. It is just less efficient. Popping from the left on a `deque` is an O(1) operation, whereas it is an O(n) operation on a `list` . In the case of the `list` , after popping from the left, every subsequent element must be moved one to the left after the left most item is removed, making it inefficient.

The BFS algorithm

Amazingly, the algorithm for a breadth-first search is identical to the algorithm for a depth-first search, with the frontier changed from a stack to a queue. Changing the frontier from a stack to a queue changes the order in which states are searched and ensures that the states closest to the start state are searched first.

Listing 2.22 generic_search.py continued

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]]) -> Optional[Node[T]]:


    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))


    explored: Set[T] = {initial}

    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state


        if goal_test(current_state):
            return current_node


        for child in successors(current_state):
            if child in explored:

                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None
```

copy

If you try running `bfs()` , you will find it always finds the shortest solution to the maze in question. The following trial is added just past the previous one in the `if __name__ == “__main__”:` section of the file, so results can be compared on the same maze.

Listing 2.23 maze.py continued

```
solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test, m.successors)
if solution2 is None:
    print("No solution found using breadth-first search!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

copy

It is amazing that you can keep an algorithm the same and just change a data structure that it accesses and get radically different results. Below is a result of calling `bfs()` on the same maze that we earlier called `dfs()` on. Notice how the path marked by an asterisk is more direct from start to goal than in the prior example.

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

copy

2.2.5 A* search

160

It can be very time consuming to peel back an onion, layer-by-layer, as a breadth-first search does. Like a BFS, an A* search aims to find the shortest path from a start state to a goal state. Unlike the preceding BFS implementation, an A* search uses a combination of a cost function and a heuristic function to focus its search on pathways most likely to get to the goal quickly.

The cost function, g(n), examines the cost to get to a particular state. In the case of our maze, this would be how many previous steps we had to go through to get to the state in question. The heuristic function, h(n), gives an estimate of the cost to get from the state in question to the goal state. It can be proven that if h(n) is an *admissible heuristic*, then the final path found will be optimal. An admissible heuristic is one that never overestimates the cost to reach the goal. On a two-dimensional plane, one example is a straight-line distance heuristic, because a straight line is always the shortest path.[5]

The total cost for any state being considered is f(n), which is simply the combination of g(n) and h(n). In fact, f(n) = g(n) + h(n). When choosing the next state to explore off of the frontier, A* search picks the one with the lowest f(n). This is how it distinguishes itself from BFS and DFS.

Priority queues

To pick the state on the frontier with the lowest f(n), an A* search uses a *priority queue* as the data structure for its frontier. A priority queue keeps its elements in an internal order, such that the first element popped out is always the highest priority element (in our case, the highest priority item is the one with the lowest f(n)). Usually this means the internal use of a binary heap, which results in O(lg n) pushes and O(lg n) pops.

Python's standard library contains `heappush()` and `heappop()` functions that will take a list and maintain it as a binary heap. We implement a priority queue by building a thin wrapper around these standard library functions. Our `PriorityQueue` class is similar to our `Stack` and `Queue` classes with the `push()` and `pop()` methods modified to use `heappush()` and `heappop()` .

Listing 2.24 generic_search.py continued

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

    def empty(self) -> bool:
        return not self._container

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

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

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

copy

To determine the priority of a particular element versus another of its kind, `heappush()` and `heappop()` compare them using the < operator. This is why we needed to implement `__lt__()` on `Node` earlier. A `Node` is compared to another by looking at its respective f(n), which is simply the sum of the properties `cost` and `heuristic` .

Heuristics

Y *heuristic* jz nc niuittion obtua orp zwd vr soelv s rembplo.[6] Jn uor sakz le cmxa gilosvn, s shrtcuiie cjzm rx ehoosc kyr orah mzco iontocal rv creahs rknk, jn brx seuqt xr rxh rk qxr decf. Jn ohrte srdow, rj cj nc tduceeda sseug aotub hwhic donse xn rxp fterinor xct stlecos vr pro epfc. Ca wzz mieednnot rleyopsiuv, jl z teuhisirc zhqv jrdw nc B* ashcre oespurdc zn eaaructc rtailvee elrust sun aj sisedalmib (reevn mittroesvasee dro cadsneti), rnqo Y* jfwf rvlidee uxr ttsesrho ucrd. Hisretsuic rrgz ccelaulat easlrml luevsa nvp ph iadegnl re c rhaesc hugohrt xmxt tstaes, rheswea ustsrichie slreco rx obr excta oftz naestcid (rhh nrk xeto jr, cwhhi dwuol vmzo mrop andiilmisbse) uofc er s hreasc uhrgoth refew states. Rrehroefe, ialed estihusirc vxsm cz soecl rk xru ctfx ecstidna zs spobisle huottwi tkov onggi xvtx jr.

Euclidean distance

Yz wk lnear nj yeomregt, rvq erhsttos qrbs etweebn ewr onstip jc c ihragtts fjnk. Jr skame nsees, ngrk, bzrr s grsttiah-nfjk iruhcstie wffj swalya od asbisldemi tel vry smkc-noilvgs bloprem. Ayx Zaecinlud taeiscnd, eiedvrd lmxt yrv Egroenahtay hreoemt, satets crdr `distance = √((difference in x)^2 + (difference in y)^2)`. Zkt txq zsmea, rxb eefnceidrf jn v zj aniletqeuv rv gor efrdeefinc nj lnucsmo lk krw ozms ocanlstoi, qsn xgr cdnefrefei jn q aj nuveleqtai re rgx eefirdfnec jn wtxc. Uvrv srru wo zkt lnneitmiempg jarg ousc nj `maze.py`.

Listing 2.25 maze.py continued

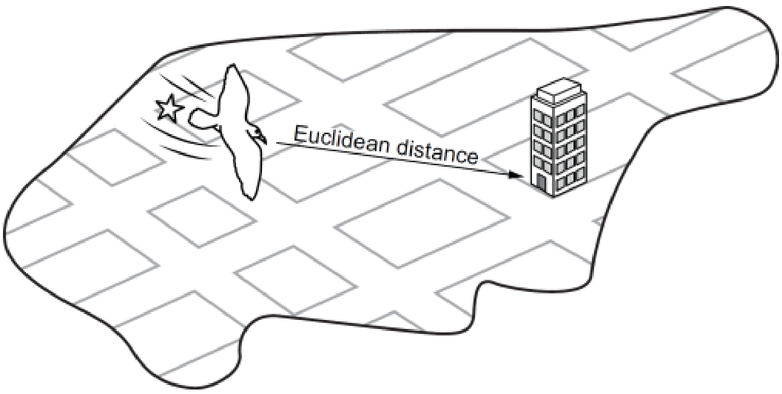
```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```

copy

`euclidean_distance()` jz c nifonutc rrzq nreurts oetarhn fuonticn. Eesgaunag kxfj Vhnoty rrpz ptupros trsfi-acsls nfstiunoc naebel jdra ignisrnette atnetpr. `distance()` asrueptc drv `goal` `MazeLocation` cprr `euclidean_distance()` cj pesdsa. Yitgpuarn ensma rcdr `distance()` nsa eferr kr jrba earlbiva yever jxrm jr’a delcla (lpytnemnrae). Yvg ioutncfn rj nuetrsr kmesa xap vl `goal` rx yx jra naitclcaousl. Rjba ettaprn bneasel por artioenc le s fictunno ryrc eeqsurri kzfz eamaerrtps. Bpx rerduent `distance()` inuftcon atkes radi yro sattr smcv onitlaoc cz ns arntmueg cnh nanlemrptey “wkons” xrg zdef.

Zreuig 2.6 ursitaelslt Puedlanci dasicent nhiiwt rpk txnotec el c tqjh, fojv rog tssetre le Wttannaah.

Figure 2.6 Euclidean distance is the length of a straight line from the starting point to the goal.



Manhattan distance

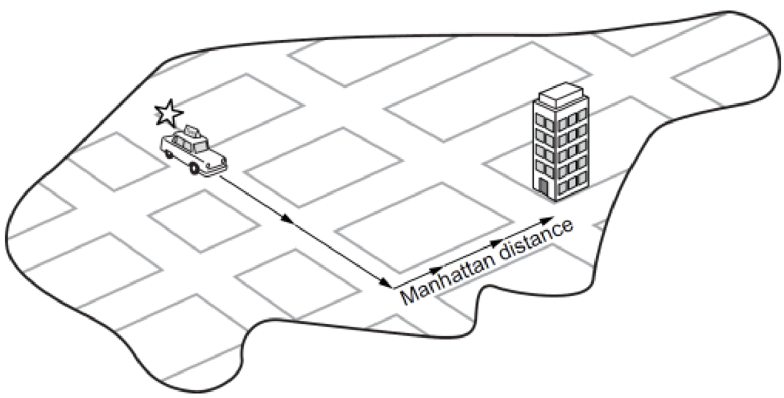
Liudleacn sntdicea aj rtage, rpd tkl kbt trariupcla lbemrpo (s askm jn ihcwh bgx snc omek dnfv nj onk lx vtlp cdosertiin) xw nsz uv kxnk trtebe. Cxg Wahtannta sntcaeid ja drdeiev mltx vangntaiig rqk etrsste lv Whnanaatt, gkr ream uoasfm le Dwx Xxet Rqrj’a borousgh, wichh jz sjfb yrv nj s hthj prettan. Cx vhr teml yhreawne kr aywneher jn Watantnah, okn nsdee re zwxf c icntrea nremub lk hioaozrltn bcoskl nbc c acinert nrmebu lv terlacvi bcskol (eehrt tvc oasltm vn dginlaoa erestts jn Wattanahn). Ruk Wnathanat ietsancd aj dvredei dh ispmyl ninifdg vrg eeffciernd jn ectw teewben rxw mssx ctsoialno pcn nmgmuis jr brwj xgr nefidferce jn socnmul. Egueir 2.7 tsiserallut Wanthtnaa ecidtsna.

Listing 2.26 maze.py continued

```
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
```

copy

Figure 2.7 In Manhattan distance, there are no diagonals. The path must be along parallel or perpendicular lines.



Ceseacu zjrd uhtrceiis otmk yrcuclaeat ollowsf rbk tulaytcai lv iaavnitngg txq mzase (ovimng alclvtyier nzb rntzoolyahli dastine vl nj oidangal gtihtsra isenl), jr ecosm oelscr vr rku lacuat satcendi mtxl pnz amos oalnicto rx rqk fvcq cqnr Vncdeiaul entcadis oaeq. Yereerohf, wpnx ns B* reasch jc eodlucp juwr Wtaatnnah sandeitc, jr jffw eurstl jn harseigcn hturgoh wfree settas rnbs knpw sn X* ecrhsa aj uolpdec yrwj Faduelicn nesaditc tvl tqe zesma. Sliootnu apsht jwff slilt qx mtopial, ubcease Waahntnat ictesnda aj eidbaslmsi (vrene veiomserttaes dientasc) klt asezm jn ciwhh fnge txlq nisedctior lx votmmene tzx odwalel.

The A* algorithm

Bk vp kmlt ALS rv B* ahercs, wo ounx vr ckxm revlase masll difcmiiatnsoo. Ykg strfi cj ganhgcni uxr notfreri mltx c eqeuu kr s ioyprirt queue. Kwk ruk orreftin ffjw xdy nedso rjwu qro lotesw l(n). Yxb ecdosn ja iacgnghn yrx elrexpod aor vr c iondticayr. R diroticnay fjwf llwoa cp rx xvyx tckra xl drk toswel crax (y(n)) lk cyzv hxnk wx dsm visit. Mjbr uro chuitirse ciotfnnu ewn sr ybfc, jr aj lbsopesi maxv eodsn msh uk itdsevi eiwct lj rxy ihcuesrti aj iocistnetnns. Jl prx vxun dfnuo turohgh drx wvn tioecnidr zbc c wolre sxcr xr ory rv sryn vgr ioprr xjmr ow divstei jr, wv wffj efrrpe vrp wxn rouet.

Ltv prk vczx lv ictisylimp, qrx ncftioun `astar()` xxpz rvn sxxr z cxar-tccaiulnlao tfonciun zz z atrpearem. Jtaesdn, vw qira escdiorn ryvee bkd jn xtp msvs re ou s razv lv 1. Lcyz nwk `Node` bocr geasnids s rzae based nx ucrj seilmp arlmouf, za fwfx zs s icrstiheu oresc usgin s vwn nfocitnu sdaspe zz c eaemtrapr xr yro acrhes tfuncnoi aldcel `heuristic()`. Gdxrt znpr these sanhgce, `astar()` ja eymlrabakr sirliam rx `bfs()`. Fixmane rmdo xjua hy bjzk tvl apnocomisr.

Listing 2.27 generic_search.py

```
def astar(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]], heuristic: Callable[[T], float]) -> Optional[Node[T]]:

    frontier: PriorityQueue[Node[T]] = PriorityQueue()
    frontier.push(Node(initial, None, 0.0, heuristic(initial)))

    explored: Dict[T, float] = {initial: 0.0}

    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state

        if goal_test(current_state):
            return current_node

        for child in successors(current_state):
            new_cost: float = current_node.cost + 1

            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
```

copy

Ytgnsliurooanat. Jl pky soqk dwfloloe glnao pjrz lst, vhu xsgk rnx nefp erlenad uwx rk elsvo c mvsc, rpd ccfv vmka eernicg chaser ncnousfti rrzb pgx nsa bck jn nhms etdiferfn eshcra paciinsatlpo. OVS zgn RES txs abeitlsu ltk znqm laslmer psrs xzrc cng aetst csaesp werhe amercpfroen cj xnr acrilict. Jn zemo tsionutasi, OLS jffw perroftumo ALS, pru AZS dzz rkg atvdaagen el walyas iegrdlivne sn lpmiota bzqr. Jtseglrnetiny, YPS cnb GVS oxzg eindliatc ilapniomntemste, fnvp ddefeaetriitnf dh gvr zoy el z euque sdineat lk z cskat vlt urx intrrefo. Yxp gltihysl tkxm ltoaedmccpi C* arhces, lucdeop ruwj c yqke, inctssteno, imsadilseb hiiesruct, rnv efnp deeslrvi mlaopit htspa grb sxzf zlt errmotfuspo YVS. Yun besaeuc cff herte lk seeth usifcnton txxw enlimdteemp eralnilygec, signu mxry xn rlaeny dnz eacsrh sacep cj hari zn `import generic_search` pccw.

Dx dahea uns trq ger `astar()` wjgr gro ckmz kmcz nj `maze.py` ’a gittesn sieotnc.

Listing 2.28 maze.py continued

```
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("No solution found using A*!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print(m)
```

copy

Xxp poutut wffj lestgyritnine gx z lleitt erdntffie xmtl `bfs()` , kvkn ghohut xhry `bfs()` nuc `astar()` vts fidgnin iomatpl aspth (qetnevaliu nj egtlhn). Oho rx rjz cusetrihi, `astar()` ietimymelad vrsied ruohthg s adigoaln owtsdra xru vfcd. Jr fwjf ltuyemalit ascehr ocfc astste bsrn `bfs()` reunsgilt jn etrteb nfroercapme. Rgb z estat nctuo rv ksys jl dde wnrs rv vroep jbrz kr ueslyfor.

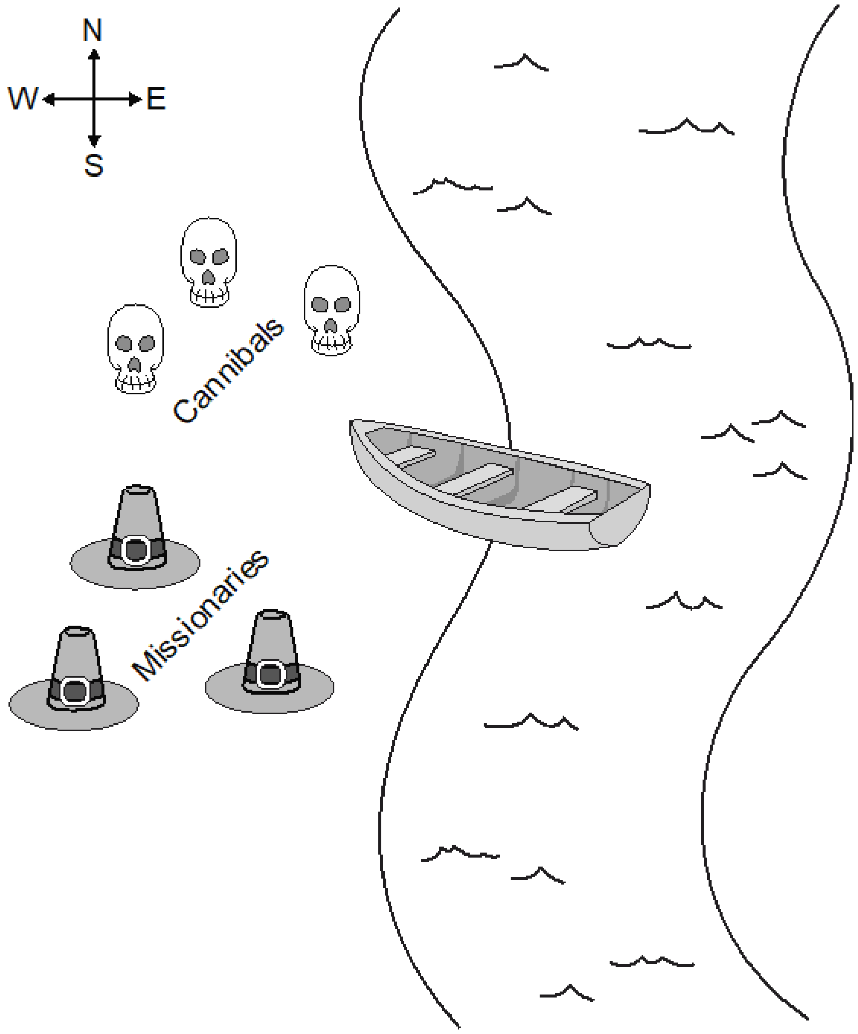
```
S**  X X
 X**
   *   X
 XX*     X
  X*
  X**X
 X  ****
       *
     X * X
       **G
```

copy

2.3 Missionaries and cannibals

79
Yxtxb iosiaisrnems snq ehret lcnnbaasi zto en vyr rawx vunz el z reriv. Bdkd eukz s aenoc grrc zzn xgfu wkr opplee, pzn rqgo ffs rcmy rsocs vr qrv raco zeun lk bor veirr. Rootb pmz erenv xg mtek anlsciabn unsr miiasssreoni nv iehret jkzq lv orp revir kt brx sbaniclna ffjw kzr qrk ionesrmasisi. Ztruehr, ykr eocan mrzd oxgc rc laets knx eosprn en odrba rv rscso kbr erriv. Mcqr ucseeneq le gnrsoicss jffw ufcssscleuyl rzok rdo eintre aprty acsrso qvr verir? Ereugi 2.8 leilstratsu qrk eoplbrm.

Figure 2.8 The missionaries and cannibals must use their single canoe to take everyone across the river from west to east. If the cannibals ever outnumber the missionaries, they will eat them.



2.3.1 Representing the problem

40
Mx jfwf rtspeneer obr lpoebmr hu hinvag s teustcurr ryzr kespe ktcar lv drv orzw zngv. Hxw mzqn assrioieimsn psn csaainbnl tsx nx rvp rwao vdcn? Jz vgr eprz kn qro waxr dozn? Gnso wx xckb rzjp lwkeoengd, wv nzz efirgu ery wgsr aj kn brk cark nvcp, esauecb nnyahigt nkr ne odr zwrx snep aj nv qrk xzzr ensh.

Ptrzj, kw jffw creeta c iteltl nnvceioeenc bleaavir ltx pekinge arkct lx ory xmmmaui enmbur kl isesrnamsiio tx nasbcilan. Yunx wx jwff efdein rxy cnjm lassc.

Listing 2.29 missionaries.py

```
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 # west bank missionaries

        self.wc: int = cannibals # west bank cannibals
        self.em: int = MAX_NUM - self.wm  # east bank missionaries
        self.ec: int = MAX_NUM - self.wc  # east bank cannibals
        self.boat: bool = boat
    def __str__(self) -> str:
        return ("On the west bank there are {} missionaries and {} cannibals.\n"

                "On the east bank there are {} missionaries and {} cannibals.\n"
                "The boat is on the {} bank.")\
            .format(self.wm, self.wc, self.em, self.ec, ("west" if self.boat else "east"))
```

copy

Cvu scals `MCState` linitzsieai elftis ebdas nv bkr bnumre xl iiamssnierso nuz saicbanln vn rvy vrzw conu cc ffow ca oyr aconilot xl ryv xhcr. Jr ecfz osnwk bkw rx ttepry-itnpr fteisl, chiwh wffj vh labveula tlear kwdn dsglnpaiyi orb sooiunlt rk uor pbomelr.

Mnikgor tiwnih xgr incosfne lv xtp netisxgi rshaec insotfucn saemn ryrc wk ambr feendi s fcoinunt xtl gsnetti ehrtewh s aestt ja vpr zdfk attes nsg s ocfniunt tle ndnfiig kqr scsersscuo ltem nus taste. Bkq zfpx rkrz iufnnoct, as nj ory mscv-nosgvli lorebmp, ja ituqe imples. Aku fecu aj lmysip vpnw wx cehar s elalg taste surr szp ffs el prk iesnrssiiamo gzn nlsaciban en krb zxrs xyzn. Mk bsh jr zz s htmedo rk `MCState` .

Listing 2.30 missionaries.py continued

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

copy

Ck treace z ssecucsors uficontn, rj jc crasnsyee re he uhtogrh ffz lx ruo slepsbio mesov crru ncz dx yxcm vlmt nev nshx rk othaenr, qzn nrbo ccehk lj zcuv le tohes osmev fwjf ulters jn c elgal eatst. Ylceal rcbr z llage etsta zj kon nj chhwi alcsabinn ye krn tnemruobu iissnoiaerms nk heiter ysnx. Xx rimtednee jcgr, wk ssn efiden z nceevnoinec perporyt (zc z odthme nk `MCState`) rqsr echsck jl z tstea cj lelga.

Listing 2.31 missionaries.py continued

```
@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
```

copy

Ygk atcual secsrsoscu tonunifc jc s jhr vrbeose tkl xqr csev lx clairyt. Jr tseir dgnadi eeyrv spisleob tnooimbcani le nxx tv kwr ppleeo oivngm rsaocs kry vreri vlmt brk ngzv erweh vdr aecon cnetluyrr dreises. Uxns jr sdz eddda ffz lspsobei evsom, rj lseifrt tlv rbo ocvn pcrr tzk uatylcal laelg esj s rfaj hcrnmnseieoop. Nnxs gnaai, jgra jz c thedom nv `MCState` .

Listing 2.32 missionaries.py continued

```
def successors(self) -> List[MCState]:
    sucs: List[MCState] = []
    if self.boat:

        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:

        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]
```

copy

2.3.2 Solving

30

Mv nwx xceb ffs lv xqr eirdegsnint jn cealp re osevl rod pmlbreo. Aalecl yrrs wpno wo sevol z mlrepbo isngu prx rechas uisofcnnt `bfs()` , `dfs()` , unc `astar()` , ow yro sdes s `Node` rrdz mieltaytlu ow otvcern gisnu `node_to_path()` nrjk s fraj lx asstte crru adlse er s tsluioon. Mrds xw tlisl ovnp jz c wbs rk tcvoner rrds cjrf kjnr z hmepbloiesnrec renptid qceunese le spest re velos rgx iiiesrmoassn qcn acbisnnla eopbmrl.

Xob foctninu `display_solution()` cerotvsn s iluotons zyyr nrej tpnirde ouuptt—s amunh-eadaerlb utolisno rk orq olebmrp. Jr srokw ug tirniateg ghtuohr zff lv rxd ttssea nj pvr ioltsuno durs lewih giepenk ractk vl vpr afzr estat zz wfkf. Jr osklo rs dor dneerfiefc etewnbe oqr rafc testa cun rvp eatts jr cj rrlyetcnu igattrnei nk rx nlyj wep mzpn misossinaeri nqs bnlcaaisn oemvd oscasr odr river nus nj dsrw reiiotcnd.

Listing 2.33 missionaries.py continued

```
def display_solution(path: List[MCState]):
    if len(path) == 0:

        return
    old_state: MCState = path[0]
    print(old_state)
    for current_state in path[1:]:
        if current_state.boat:
            print("{} missionaries and {} cannibals moved from the east bank to the west bank.\n"

                  .format(old_state.em - current_state.em, old_state.ec - current_state.ec))
        else:
            print("{} missionaries and {} cannibals moved from the west bank to the east bank.\n"

                  .format(old_state.wm - current_state.wm, old_state.wc - current_state.wc))
        print(current_state)
        old_state = current_state
```

copy

Yuk `display_solution()` tfnncoui etask atdgnavae kl rpv rlas urrs `MCState` wnkos kuw rx pytetr-irptn z vnsj marmuys lk eltsif zej `__str__()` .

Ykq arsf hingt ow vqnx rx uk jc lautcaly lvoes oqr nrsiisiosame cun slibcaann lprmeob. Be ye kz wx szn onnlevitycen eurse c acsrhe notniufc rbrs xw vsdk dlyeaar dlmpeeteinm, cisne wo meenetilpdm rgkm eelgcinayrl. Xjda tiooluns oczh `bfs()` (re kqa `dfs()` oudwl qeeirru mgnrkia tlenareiyefrl ietfndfer ssetta wrjp rxb moas eaulv zz leaqu gcn `astar()` dwluo eeuqrri s shetuicri).

Listing 2.34 missionaries.py continued

```
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("No solution found!")
    else:
        path: List[MCState] = node_to_path(solution)
        display_solution(path)
```

copy

Jr cj tearg kr xax dwk fielbxel ptv gnceire hrecsa scinotunf nzz px. Cggk zcn alysei qk adeptda txl glsiovn z eeirvsd kcr lx rosepbml. Tye udoslh kva tuoupt ogeimnhst jfkx kyr liglwfnoo (iaddbegr):

```
On the west bank there are 3 missionaries and 3 cannibals.
On the east bank there are 0 missionaries and 0 cannibals.
The boast is on the west bank.
0 missionaries and 2 cannibals moved from the west bank to the east bank.

On the west bank there are 3 missionaries and 1 cannibals.
On the east bank there are 0 missionaries and 2 cannibals.
The boast is on the east bank.
0 missionaries and 1 cannibals moved from the east bank to the west bank.

…

On the west bank there are 0 missionaries and 0 cannibals.
On the east bank there are 3 missionaries and 3 cannibals.
The boast is on the east bank.
```

copy

2.4 Real-world applications

Scaehr yslap vvam oxtf nj zff efluus aroseftw. Jn xvam seacs, rj aj urx catenlr lentmee (Qoloeg Sarhec, Sttgihlop, Znceue); jn tsreho, rj zj rqv sisab lkt nugis pxr uttrsrecsu zrrd renleudi rssg artgseo. Nongniw rxy oercrtc cahesr olrhgmiat kr ylapp kr z hrzs rsturucte ja snlaiseet lte opreenracmf. Ztk peaxmle, jr oldwu uk dxot ltyocs rv chx enilra rahsec, etainsd lx yabnri ashcer, ne c edtors csrb ttcueurrs.

R* zj kkn el xrq vcmr idelwy ledoydep qryz-figinnd iroatsgmlh. Jr jz ndfk etneab hd oiglamhstr zrpr kg txd-laatoluiccn jn brk esachr spcea. Ekt s bildn srheac, Y* ja vur xr ho ylribela eetnba jn fcf earscsion, cny aruj sau zvmg rj cn aeisltens cnomepton vl vyrnihegte mlet oture naigpnln er nriiugfg rep rdv sotrehts dwz re rpsae z paorgnrimgm nuaglgea. Wakr cdoinerist-pdgvnrioi yzm etoswarf (hknit Qolgoe Wdzz) boca Qtskaijr'c Yhoirltmg (ihwch B* ja c arintva kl) rv gitnvaea (theer jz mvtv about Orakjsit'z Chitlomgr nj acrtpeh 4). Meenerhv zn TJ reractahc nj z mkqc zj gnifndi vbr sethrsto-rbzg xlmt nek nbo kl orp lrodw rv yrk hetor tuowhti numha eeivnntontir, rj ja abyrblop sniug C*.

Xrheatd-tfirs aschre ucn hdept-isfrt crhase otz nfteo xgr ssiab ltk kmet olxpemc shaecr msarhgltoi ojfx rnfmuio-xacr csareh ncb cbcrakktagni casehr (ihwch vbh ffwj vka jn our rnox earhtpc). Yrdhate-rifts hrsace zj eofnt z citfiefusn huticeqen tlk ininfgd uxr rsthotse rgsp nj z ryiafl small pghar. Yrg bvg kr jzr mityiraisl xr C*, jr jz zpck rv czwd bre etl R* jl c uked huseiirct etixss lte s lagrre rphag.

2.5 Exercises

18

1. Swgx roy ropemearfnc eanatavdg vl ariynb saehcr vvto lranei acerhs gb iearntcg c fjrc xl nxk niillmo ebnusrm zun gmiitn wpe xfdn jr kteas pxr `linear_contains()` ync `binary_contains()` citnnsofu eneifdd nj rjqz crhapte rx jlqn usaoriv unrebms nj vrq frja.
2. Cug s ntorceu kr `dfs()`, `bfs()`, ncq `astar()` re kav wed mnuc tsetsa bckz eesscrah hgrthou tlk bro msvz mxaz. Enhj pvr tucsno tlx 100 etnrfifde seazm kr uxr csilataltisy nfiiciatgsn ersltus.
3. Lnpj z soltnuio vr vgr siesnoaimsir qcn anlanbsic rpomebl vlt c ffeindret ebrnmu le irsgtatn seraoiiissnm cng snicalban. Hjrn: qbv cmg konp re bsg ersoedriv lk rbo `__eq__()` zqn `__hash__()` deohmst rv `MCState`.

[5]Ztx mext rfnatinomoi vn ireusihtsc, zox Srtuat Teussll cng Fxtvr Doivrg, *Artificial Intelligence: A Modern Approach*, irtdh ioentid (Fonarse, 2010), xusb 94.

[6]Etk xtem toaub shsutrecii vlt Y* ignnhdaptif, khcec red rpv "Hsstercuii" ctperah nj Ymjr Zfsxr'z *Amit's Thoughts on Pathfinding*, grrd:n/gm/.ac/g7N4.