Classic Computer Science Problems in Swift

Essential techniques for practicing programmers

David Kopec

SAMPLE CHAPTER
Chapter 2
brief contents

1 ■ Small problems 5
2 ■ Search problems 22
3 ■ Constraint-satisfaction problems 47
4 ■ Graph problems 65
5 ■ Genetic algorithms 95
6 ■ K-means clustering 113
7 ■ Fairly simple neural networks 129
8 ■ Miscellaneous problems 157
“Search” is such a broad term that this entire book could be called “Classic Search Problems in Swift.” 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

![DNA structure](image)

**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.
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.

### 2.1.1 Storing DNA

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

```swift
enum Nucleotide: Character, Comparable {
    case A = "A", C = "C", G = "G", T = "T"
}
```

Nucleotide needs to implement the `Comparable` interface so that Nucleotides can be put in order. An entity that implements `Comparable` must override the `<` operator. This can be done either as a freestanding function, or as a static method inside the `Comparable` entity.

Here, we implement `<` as a freestanding function by comparing one Nucleotide’s raw `Character` value against another’s. `Character` has a built-in alphabetical ordering.

```swift
func <(lhs: Nucleotide, rhs: Nucleotide) -> Bool {
    return lhs.rawValue < rhs.rawValue
}
```

Codons can be defined as a tuple of three Nucleotides. And finally, a gene may be defined as an array of Codons.

```swift
typealias Codon = (Nucleotide, Nucleotide, Nucleotide)
typealias Gene = [Codon]
```

**NOTE** Although we will later need to compare one Codon to another, we do not need to define the `<` operator for Codon. This is because Swift 2.2 introduced a generic implementation of `<` for any tuple type that contains elements of type `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 `geneSequence`.

```swift
let geneSequence = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

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

```swift
func stringToGene(_ s: String) -> Gene {
    var gene = Gene()
    for i in stride(from: 0, to: s.count, by: 3) {
        guard (i + 2) < s.count else { return gene }
        if let n1 = Nucleotide.init(rawValue: s[s.index(s.startIndex, offsetBy: i)]), let n2 = Nucleotide.init(rawValue: s[s.index(s.startIndex, offsetBy: i + 1)]), let n3 = Nucleotide.init(rawValue: s[s.index(s.startIndex, offsetBy: i + 2)]) {
            gene.append(Codon(n1, n2, n3))
        }
    }
    return gene
}
```
stringToGene() continually goes through the provided String and converts its next three characters into Codons 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 guard 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.

stringToGene() can be used to convert the String geneSequence into a Gene.

var gene = stringToGene(geneSequence)

### 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.

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 gene and a Codon called acg.

```swift
func linearContains(_: array: Gene, item: Codon) -> Bool {
    for element in gene where item == element {
        return true
    }
    return false
}

let acg: Codon = (.A, .C, .G)
linearContains(gene, item: acg)
```

**NOTE** The built-in Swift method on Sequence called `contains()` does a linear search and returns `true` if the element in question is found or `false` if it is not. It should be preferred to writing your own linear search function in most circumstances.
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 Swift Array 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 an Array of alphabetically sorted words like ["cat", "dog", "kangaroo", "llama", "rabbit", "rat", "zebra"] and we are searching for the word "rat":"n

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.

A binary search continually reduces the search space by half, so it has a worst-case runtime of $O(\log n)$. 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 $O(n \log n)$ 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 an Array.
Let’s walk through this function line by line.

```swift
var low = 0
var high = array.count - 1

while low <= high {
    let mid = (low + high) / 2
    if array[mid] < item {
        low = mid + 1
    } else if array[mid] > item {
        high = mid - 1
    } else {
        return true
    }
}
return false
```

We start by looking at a range that encompasses the entire array.

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 array.

```swift
let mid = (low + high) / 2
```

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

```swift
if array[mid] < item {
    low = mid + 1
}
```

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.

```swift
else if array[mid] > item {
    high = mid - 1
}
```

Similarly, we halve in the other direction when the element we are looking for is less than the middle element.
else {
    return true
}

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:

```swift
let sortedGene = gene.sorted(by: <)
binaryContains(sortedGene, item: acg)
```

### 2.1.4 A generic example

The functions `linearContains()` and `binaryContains()` can be generalized to work with any type that implements `Comparable`. These generalized versions are nearly identical to the versions you saw before, with only their type signatures changed.

**NOTE** As of Swift 4.0, tuples cannot be made to explicitly implement `Comparable`, so our prior types `Gene` and `Codon` cannot be used with these generic implementations.

```swift
func linearContains<T: Equatable>(_ array: [T], item: T) -> Bool {
    for element in array where item == element {
        return true
    }
    return false
}

func binaryContains<T: Comparable>(_ array: [T], item: T) -> Bool {
    var low = 0
    var high = array.count - 1
    while low <= high {
        let mid = (low + high) / 2
        if array[mid] < item {
            low = mid + 1
        } else if array[mid] > item {
            high = mid - 1
        } else {
            return true
        }
    }
    return false
}
```

Now you can try doing searches on other types of data.

```swift
linearContains([1,5,15,15,15,15,15], item: 5)
binaryContains(["a", "d", "e", "f", "g"], item: "f")
```
2.2 **Maze solving**

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 array of Cell. A Cell is an enum with raw Character values where 0 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.

```swift
// A Cell represents the status of a grid location in the maze
enum Cell: Character {
    case Empty = "O"
    case Blocked = "X"
    case Start = "S"
    case Goal = "G"
    case Path = "P"
}

typealias Maze = [[Cell]]
```

### 2.2.1 Generating a random maze

The maze that is generated should be fairly sparse so that there is almost always a path from a given starting node to a given ending node (this is for testing our algorithms, after all). We’ll let the caller of `generateMaze()` decide on the exact sparseness. When a random number beats the threshold of the sparseness parameter in question, we’ll 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.

```swift
srand48(time(nil)) // seed random number generator

// sparseness is the approximate percentage of walls represented
// as a number between 0 and 1
func generateMaze(rows: Int, columns: Int, sparseness: Double) -> Maze {
    // initialize maze full of empty spaces
    var maze: Maze = Maze(repeating: [Cell](repeating: .Empty, count: columns), count: rows)
    // put walls in
    for row in 0..<rows {
        for col in 0..<columns {
            if drand48() < sparseness { // chance of wall
                maze[row][col] = .Blocked
            }
        }
    }
    return maze
}
```
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.

```
func printMaze(_ maze: Maze) {
    for i in 0..<maze.count {
        print(String(maze[i].map{ $0.rawValue })))
    }
}
```

Go ahead and test these maze functions.

```
var maze = generateMaze(rows: 10, columns: 10, sparseness: 0.2)
printMaze(maze)
```

### Miscellaneous maze minutiae

We’ll need a way to refer to an individual location in the maze. This could be a tuple of row and column, but later we will want to store a maze location in data structures that require their keys to be `Hashable`. Instead, therefore, we will define a custom struct for maze locations (tuples do not conform and cannot be made to conform to `Hashable`). All `Hashable` conforming types must also implement the `==` operator.

```
struct MazeLocation: Hashable {
    let row: Int
    let col: Int
    var hashValue: Int { return row.hashValue ^ col.hashValue }  
}
```

```
func ==(lhs: MazeLocation, rhs: MazeLocation) -> Bool {
    return lhs.row == rhs.row && lhs.col == rhs.col
}
```

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’ll arbitrarily define the goal as always being at location 9, 9 for now.

```
let goal = MazeLocation(row: 9, col: 9)
func goalTest(ml: MazeLocation) -> Bool {
    return ml == goal
}
```

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 a `successorsForMaze()` function that returns an appropriate `successors()` function for the `Maze` in question.
func successorsForMaze(_: maze: Maze) -> (MazeLocation) -> [MazeLocation] {
    func successors(ml: MazeLocation) -> [MazeLocation] { //no diagonals
        var newMLs: [MazeLocation] = [MazeLocation]()
        if (ml.row + 1 < maze.count) && (maze[ml.row + 1][ml.col] != .Blocked) {
            newMLs.append(MazeLocation(row: ml.row + 1, col: ml.col))
        }
        if (ml.row - 1 >= 0) && (maze[ml.row - 1][ml.col] != .Blocked) {
            newMLs.append(MazeLocation(row: ml.row - 1, col: ml.col))
        }
        if (ml.col + 1 < maze[0].count) && (maze[ml.row][ml.col + 1] != .Blocked) {
            newMLs.append(MazeLocation(row: ml.row, col: ml.col + 1))
        }
        if (ml.col - 1 >= 0) && (maze[ml.row][ml.col - 1] != .Blocked) {
            newMLs.append(MazeLocation(row: ml.row, col: ml.col - 1))
        }
        return newMLs
    }
    return successors
}

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 an array that it ultimately returns to the caller.

The pattern of successors(), a function returning a function, is unusual in Objective-C and in most pure object-oriented programming languages, but it is common in functional languages. The inner function captures data from the outer function. In this case, successors() captures maze from successorsForMaze(). Such a pattern can sometimes be confusing to implement, but ultimately it can offer more convenience for the user of an API. An alternative, equally convenient pattern would be to create a Maze class (instead of using a raw typealias) and add a successors() method to it. Neither approach is inherently wrong or right. The fact that both are possible in Swift shows its flexibility and that it straddles both the functional and object-oriented worlds.

### 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.

**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.
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.

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 linked list. We will implement our stack on top of Swift’s `Array` 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 `isEmpty` property to check if the stack has any more items in it.\(^1\)

```swift
public class Stack<T> {
    private var container: [T] = [T]()
    public var isEmpty: Bool { return container.isEmpty }
    public func push(thing: T) { container.append(thing) }
    public func pop() -> T { return container.removeLast() }
}
```

---

\(^1\) These examples are based on prior code I wrote for the SwiftGraph open source project: https://github.com/davecom/SwiftGraph.
Note that implementing a stack using a Swift `Array` is as simple as always appending items onto its right end, and always removing items from its extreme right end. The `removeLast()` method on `Array` will fail if there are no longer any items in the array, 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 as being `Comparable` and `Hashable`, so we can reuse it later in the A* algorithm.

```swift
class Node<T>: Comparable, Hashable {
    let state: T
    let parent: Node?
    let cost: Float
    let heuristic: Float

    init(state: T, parent: Node?, cost: Float = 0.0, heuristic: Float = 0.0) {
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic
    }

    var hashValue: Int { return Int(cost + heuristic) }
}
```

```swift
func < <T>(lhs: Node<T>, rhs: Node<T>) -> Bool {
    return (lhs.cost + lhs.heuristic) < (rhs.cost + rhs.heuristic)
}
```

```swift
func == <T>(lhs: Node<T>, rhs: Node<T>) -> Bool {
    return lhs === rhs
}
```

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 visited. 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 visited, 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.
func dfs<StateType, Hashable>(initialState: StateType, goalTestFn: (StateType) -> Bool, successorFn: (StateType) -> [StateType]) -> Node<StateType>? {
    // frontier is where we've yet to go
    let frontier: Stack<Node<StateType>> = Stack<Node<StateType>>()
    frontier.push(Node(state: initialState, parent: nil))
    // explored is where we've been
    var explored: Set<StateType> = Set<StateType>()
    explored.insert(initialState)

    // keep going while there is more to explore
    while !frontier.isEmpty {
        let currentNode = frontier.pop()
        let currentState = currentNode.state
        // if we found the goal, we're done
        if goalTestFn(currentState) { return currentNode }
        // check where we can go next and haven't explored
        for child in successorFn(currentState) where
            !explored.contains(child) {
                explored.insert(child)
                frontier.push(Node(state: child, parent: currentNode))
            }
    }
    return nil // never found the goal
}

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.

func nodeToPath<StateType>(_ node: Node<StateType>) -> [StateType] {
    var path: [StateType] = [node.state]
    var node = node // local modifiable copy of reference
    // work backwards from end to front
    while let currentNode = node.parent {
        path.insert(currentNode.state, at: 0)
        node = currentNode
    }
    return path
}

For display purposes, it will be useful to mark up the maze with the successful path,
the start state, and the goal state.

func markMaze(_ maze: inout Maze, path: [MazeLocation], start: MazeLocation,
              goal: MazeLocation) {
    for ml in path {
        maze[ml.row][ml.col] = .Path
    }
    maze[start.row][start.col] = .Start
    maze[goal.row][goal.col] = .Goal
}
NOTE inout indicates that the original object passed as maze will be modified by markMaze() instead of simply being copied into a temporary variable within markMaze() and forgotten about. At optimization time, inout is analogous to “call by reference” in other programming languages. To be clear—markMaze() modifies the original maze it is passed. The changes that are made to that maze will persist after the function ends. At call time, inout arguments are passed with a preceding ampersand, &.

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

```swift
let start = MazeLocation(row: 0, col: 0)

if let solution = dfs(initialState: start, goalTestFn: goalTest, successorFn: successorsForMaze(maze)) {
    let path = nodeToPath(solution)
    markMaze(&maze, path: path, start: start, goal: goal)
    printMaze(maze)
}
```

A successful solution will look something like this:

```
SPX0000XOO
OPPPPPPPPPO
X0000000PO
0000XPPPPX
OXX0XPXOXO
OXPPPPOOOO
PPXX0XXX0X
P000X0000X
PPPXXXXPPP
000X0000OG
```

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.

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

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 Swift Array) is almost identical to our implementation of Stack, with the only change being the removal of elements from the left end of the Array instead of the right end. The elements on the left end are the oldest elements still in the Array (in terms of arrival time), so they are the first elements popped.²

```swift
public class Queue<T> {
    private var container: [T] = [T]()
    public var isEmpty: Bool { return container.isEmpty }
    public func push(thing: T) { container.append(thing) }
    public func pop() -> T { return container.removeFirst() }
}
```

**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.

```swift
func bfs<StateType: Hashable>(initialState: StateType, goalTestFn: (StateType) -> Bool, successorFn: (StateType) -> [StateType]) -> Node<StateType>? {
    // frontier is where we've yet to go
    let frontier: Queue<Node<StateType>> = Queue<Node<StateType>>()
    frontier.push(Node(state: initialState, parent: nil))
    // explored is where we've been
    var explored: Set<StateType> = Set<StateType>()
    explored.insert(initialState)
    // keep going while there is more to explore
    while !frontier.isEmpty {
        let currentNode = frontier.pop()
        let currentState = currentNode.state
        // if we found the goal, we're done
        if goalTestFn(currentState) { return currentNode }
        // check where we can go next and haven't explored
        for child in successorFn(currentState) where !explored.contains(child) {
            explored.insert(child)
            frontier.push(Node(state: child, parent: currentNode))
        }
    }
    return nil // never found the goal
}
```

If you try running `bfs()` , you will find it always finds the shortest solution to the maze in question.

---

² These examples are based on prior code I wrote for the SwiftGraph open source project: https://github.com/davecom/SwiftGraph.
2.2.5 A* search

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.)³

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.

Although the standard libraries of many modern programming languages contain a built-in priority queue, Swift’s does not. We will not implement a priority queue from scratch. Instead we will utilize the open source project SwiftPriorityQueue, which I built.⁴

To determine the priority of a particular element versus another of its kind, SwiftPriorityQueue requires that the type of its elements implements the Swift standard library protocol `Comparable`. This is why the Node class was defined as `Comparable` and

---


⁴ These examples are based on prior code I wrote for the SwiftPriorityQueue open source project: [https://github.com/davecom/SwiftPriorityQueue](https://github.com/davecom/SwiftPriorityQueue).
therefore had to implement the < operator. 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**

A heuristic is an intuition about the way to solve a problem. In the case of maze solving, a heuristic aims to choose the best maze location to search next, in the quest to get to the goal. In other words, it is an educated guess about which nodes on the frontier are closest to the goal. As was mentioned previously, if a heuristic used with an A* search produces an accurate relative result and is admissible (never overestimates the distance), then A* will deliver the shortest path. Heuristics that calculate smaller values end up leading to a search through more states, whereas heuristics closer to the exact real distance (but not over it, which would make them inadmissible) lead to a search through fewer states. Therefore, ideal heuristics come as close to the real distance as possible without ever going over it.

**Euclidean distance**

As we learn in geometry, the shortest path between two points is a straight line. It makes sense, then, that a straight-line heuristic will always be admissible for the maze-solving problem. The Euclidean distance, derived from the Pythagorean theorem, states that \( \text{distance} = \sqrt{((\text{difference in x})^2 + (\text{difference in y})^2)} \). For our mazes, the difference in x is equivalent to the difference in columns of two maze locations, and the difference in y is equivalent to the difference in rows.

```swift
func euclideanDistance(ml: MazeLocation) -> Float {
    let xdist = ml.col - goal.col
    let ydist = ml.row - goal.row
    return sqrt(Float((xdist * xdist) + (ydist * ydist)))
}
```

Figure 2.6 illustrates Euclidean distance within the context of a grid, like the streets of Manhattan.

---

5 For more about heuristics for A* pathfinding, check out the “Heuristics” chapter in Amit Patel’s *Amit’s Thoughts on Pathfinding*, [http://mng.bz/z7O4](http://mng.bz/z7O4).
**MANHATTAN DISTANCE**

Euclidean distance is great, but for our particular problem (a maze in which you can move only in one of four directions) we can do even better. The Manhattan distance is derived from navigating the streets of Manhattan, the most famous of New York City’s boroughs, which is laid out in a grid pattern. To get from anywhere to anywhere in Manhattan, one needs to walk a certain number of horizontal blocks and a certain number of vertical blocks (there are almost no diagonal streets in Manhattan). The Manhattan distance is derived by simply finding the difference in rows between two maze locations and summing it with the difference in columns. Figure 2.7 illustrates Manhattan distance.

```swift
func manhattanDistance(ml: MazeLocation) -> Float {
    let xdist = abs(ml.col - goal.col)
    let ydist = abs(ml.row - goal.row)
    return Float(xdist + ydist)
}
```

![Figure 2.7 In Manhattan distance, there are no diagonals. The path must be along parallel or perpendicular lines.](image)

Because this heuristic more accurately follows the actuality of navigating our mazes (moving vertically and horizontally instead of in diagonal straight lines), it comes closer to the actual distance from any maze location to the goal than Euclidean distance does. Therefore, when an A* search is coupled with Manhattan distance, it will result in searching through fewer states than when an A* search is coupled with Euclidean distance for our mazes. Solution paths will still be optimal, because Manhattan distance is admissible (never overestimates distance) for mazes in which only four directions of movement are allowed.

**THE A* ALGORITHM**

To go from BFS to A* search, we need to make several small modifications. The first is changing the frontier from a queue to a priority queue. Now the frontier will pop nodes with the lowest f(n). The second is changing the explored set to a dictionary. A
dictionary will allow us to keep track of the lowest cost (g(n)) of each node we may visit. With the heuristic function now at play, it is possible some nodes may be visited twice if the heuristic is inconsistent. If the node found through the new direction has a lower cost to get to than the prior time we visited it, we will prefer the new route.

For the sake of simplicity, the function `astar()` does not take a cost-calculation function as a parameter. Instead, we just consider every hop in our maze to be a cost of 1. Each new `Node` gets assigned a cost based on this simple formula, as well as a heuristic score using a new function passed as a parameter to the search function called `heuristicFn()`. Other than these changes, `astar()` is remarkably similar to `bfs()`.

Examine them side by side for comparison.

```swift
func astar<StateType: Hashable>(initialState: StateType, goalTestFn: (StateType) -> Bool, successorFn: (StateType) -> [StateType], heuristicFn: (StateType) -> Float) -> Node<StateType>? {
    // frontier is where we've yet to go
    var frontier: PriorityQueue<Node<StateType>> = PriorityQueue<Node<StateType>>(ascending: true, startingValues: [Node(state: initialState, parent: nil, cost: 0, heuristic: heuristicFn(initialState))])
    // explored is where we've been
    var explored = Dictionary<StateType, Float>()
    explored[initialState] = 0
    // keep going while there is more to explore
    while let currentNode = frontier.pop() {
        let currentState = currentNode.state
        // if we found the goal, we're done
        if goalTestFn(currentState) { return currentNode }
        // check where we can go next and haven't explored
        for child in successorFn(currentState) {
            let newcost = currentNode.cost + 1 //1 assumes a grid, there
            if (explored[child] == nil) || (explored[child]! > newcost) {
                explored[child] = newcost
                frontier.push(Node(state: child, parent: currentNode, cost: newcost, heuristic: heuristicFn(child)))
            }
        }
    }
    return nil // never found the goal
}
```

Congratulations. If you have followed along this far, you have not only learned how to solve a maze, but also some generic search functions that you can use in many different search applications. DFS and BFS are suitable for many smaller data sets and state spaces where performance is not critical. In some situations, DFS will outperform BFS, but BFS has the advantage of always delivering an optimal path. Interestingly,

---

6 These examples are based on prior code I wrote for the SwiftPriorityQueue open source project: [https://github.com/davecom/SwiftPriorityQueue](https://github.com/davecom/SwiftPriorityQueue).
BFS and DFS have identical implementations, only differentiated by the use of a queue instead of a stack for the frontier. The slightly more complicated A* search, coupled with a good, consistent, admissible heuristic, not only delivers optimal paths but also far outperforms BFS.

Go ahead and try out `astar()`.

```swift
var maze3 = generateMaze(rows: 10, columns: 10, sparseness: 0.2)
if let solution = astar(initialState: start, goalTestFn: goalTest,
    successorFn: successorsForMaze(maze3), heuristicFn: manhattanDistance) {
    let path = nodeToPath(solution)
    markMaze(&maze3, path: path, start: start, goal: goal)
    printMaze(maze3)
}
```

### 2.3 Missionaries and cannibals

Three missionaries and three cannibals are on the west bank of a river. They have a canoe that can hold two people, and they all must cross to the east bank of the river. There may never be more cannibals than missionaries on either side of the river or the cannibals will eat the missionaries. Further, the canoe must have at least one person on board to cross the river. What sequence of crossings will successfully take the entire party across the river? Figure 2.8 illustrates the problem.

![Missionaries and cannibals](image)

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

We will represent the problem by having a structure that keeps track of the west bank. How many missionaries and cannibals are on the west bank? Is the boat on the west bank? Once we have this knowledge, we can figure out what is on the east bank, because anything not on the west bank is on the east bank.

First, we will create a little convenience variable for keeping track of the maximum number of missionaries or cannibals. Then we will define the main structure.

```swift
let maxNum = 3 // max number of missionaries or cannibals

struct MCState: Hashable, CustomStringConvertible {
    let missionaries: Int
    let cannibals: Int
    let boat: Bool

    var hashValue: Int { return missionaries * 10 + cannibals + (boat ? 1000 : 2000) }

    var description: String {
        let wm = missionaries // west bank missionaries
        let wc = cannibals // west bank cannibals
        let em = maxNum - wm // east bank missionaries
        let ec = maxNum - wc // east bank cannibals
        var description = "On the west bank there are \(wm) missionaries and \(wc) cannibals.
        "
        description += "On the east bank there are \(em) missionaries and \(ec) cannibals.\n        "
        description += "The boat is on the \(boat ? "west" : "east") bank.\n        "
        return description
    }
}
```

func ==(lhs: MCState, rhs: MCState) -> Bool {
    return lhs.hashValue == rhs.hashValue
}
```

The struct `MCState` implements `Hashable` and `CustomStringConvertible`. It implements `Hashable` because we want to be able to use it within the framework of our existing search functions and because we want to be able to distinguish one state from another. It implements `CustomStringConvertible` because we want to be able to print out a nicely formatted description of a given state in our program.

Working within the confines of our existing search functions means that we must define a function for testing whether a state is the goal state and a function for finding the successors from any state. The goal test function, as in the maze-solving problem, is quite simple. The goal is simply when there are no longer any people on the west bank of the river.

```swift
func goalTestMC(state: MCState) -> Bool {
    return state == MCState(missionaries: 0, cannibals: 0, boat: false)
}
```
To create a successors function, it is necessary to go through all of the possible moves that can be made from one bank to another, and then check if each of those moves will result in a legal state. Recall that a legal state is one in which cannibals do not outnumber missionaries on either bank. To determine this, we can define a convenience function that checks if a state is legal.

```swift
func isLegalMC(state: MCState) -> Bool {
    let wm = state.missionaries // west bank missionaries
    let wc = state.cannibals // west bank cannibals
    let em = maxNum - wm // east bank missionaries
    let ec = maxNum - wc // east bank cannibals
    // check there's not more cannibals than missionaries
    if wm < wc && wm > 0 { return false }
    if em < ec && em > 0 { return false }
    return true
}
```

The actual successors function is a bit verbose for the sake of clarity. It tries adding every possible combination of one or two people moving across the river from the bank where the canoe currently resides. Once it has added all possible moves, it filters for the ones that are actually legal.

```swift
func successorsMC(state: MCState) -> [MCState] {
    let wm = state.missionaries // west bank missionaries
    let wc = state.cannibals // west bank cannibals
    let em = maxNum - wm // east bank missionaries
    let ec = maxNum - wc // east bank cannibals
    var sucs: [MCState] = [MCState]() // next states

    if state.boat { // boat on west bank
        if wm > 1 {
            sucs.append(MCState(missionaries: wm - 2, cannibals: wc, boat: !state.boat))
        }
        if wm > 0 {
            sucs.append(MCState(missionaries: wm - 1, cannibals: wc, boat: !state.boat))
        }
        if wc > 1 {
            sucs.append(MCState(missionaries: wm, cannibals: wc - 2, boat: !state.boat))
        }
        if wc > 0 {
            sucs.append(MCState(missionaries: wm, cannibals: wc - 1, boat: !state.boat))
        }
        if (wc > 0) && (wm > 0) {
            sucs.append(MCState(missionaries: wm - 1, cannibals: wc - 1, boat: !state.boat))
        }
    } else { // boat on east bank
        if em > 1 {
```
sucs.append(MCState(missionaries: wm + 2, cannibals: wc, boat: !state.boat))
}
if em > 0 {
    sucs.append(MCState(missionaries: wm + 1, cannibals: wc, boat: !state.boat))
}
if ec > 1 {
    sucs.append(MCState(missionaries: wm, cannibals: wc + 2, boat: !state.boat))
}
if ec > 0 {
    sucs.append(MCState(missionaries: wm, cannibals: wc + 1, boat: !state.boat))
}
if (ec > 0) && (em > 0){
    sucs.append(MCState(missionaries: wm + 1, cannibals: wc + 1, boat: !state.boat))
}
}
return sucs.filter{ isLegalMC(state: $0) }

2.3.2 Solving

We now have all of the ingredients in place to solve the problem. Recall that when we solve a problem using the search functions `bfs()`, `dfs()`, and `astar()`, we get back a `Node` that ultimately we convert using `nodeToPath()` into an array of states that leads to a solution. What we still need is a way to convert that array into a comprehensible printed sequence of steps to solve the missionaries and cannibals problem.

The function `printMCSolution()` converts a solution path into printed output—a human-readable solution to the problem. It works by iterating through all of the states in the solution path while keeping track of the last state as well. It looks at the difference between the last state and the state it is currently iterating on to find how many missionaries and cannibals moved across the river and in what direction.

```swift
func printMCSolution(path: [MCState]) {
    var oldState = path.first!
    print(oldState)
    for currentState in path[1..<path.count] {
        let wm = currentState.missionaries // west bank missionaries
        let wc = currentState.cannibals // west bank cannibals
        let em = maxNum - wm // east bank missionaries
        let ec = maxNum - wc // east bank cannibals
        if !currentState.boat {
            print("(oldState.missionaries - wm) missionaries and
                  \(oldState.cannibals - wc) cannibals moved from the west bank
to the east bank.")
        } else {
            print("\(maxNum - oldState.missionaries - em) missionaries
and \(maxNum - oldState.cannibals - ec) cannibals moved from
"
The `printMCSolution()` function takes advantage of the fact that `MCState` is `StringConvertible` to print out a state’s description with `print()`. The last thing we need to do is actually solve the missionaries and cannibals problem. To do so we could use any of our previously implemented search functions. This solution uses `bfs()`.

```swift
let startMC = MCState(missionaries: 3, cannibals: 3, boat: true)
if let solution = bfs(initialState: startMC, goalTestFn: goalTestMC,
    successorFn: successorsMC) {
    let path = nodeToPath(solution)
    printMCSolution(path: path)
}
```

It is great to see how flexible our generic search functions can be. They can easily be adapted for solving a diverse set of problems.

## 2.4 Real-world applications

Search plays some role in all useful software. In some cases it is the central element (Google Search, Spotlight, Lucene); in others it is the basis for using the structures that underlie data storage. Knowing the correct search algorithm to apply to a data structure is essential for performance. For example, it would be very costly to use linear search, instead of binary search, on a sorted data structure.

A* is one of the most widely deployed path-finding algorithms. It is only beaten by algorithms that do precalculation in the search space. For a blind search, A* is yet to be reliably beaten in all scenarios, and this has made it an essential component of everything from route planning to figuring out the shortest way to parse a programming language. Most directions-providing map software (think Google Maps) uses Dijkstra’s Algorithm (which A* is a variant of) to navigate (there is more about Dijkstra’s Algorithm in chapter 4). Whenever an AI character in a game is finding the shortest-path from one end of the world to the other without human intervention, it is probably using A*.

Breadth-first search and depth-first search are often the basis for more complex search algorithms like uniform-cost search and backtracking search (which you will see in the next chapter). Breadth-first search is often a sufficient technique for finding the shortest path in a fairly small graph. But due to its similarity to A*, it is easy to swap out for A* if a good heuristic exists for a larger graph.
2.5 Exercises

1. Show the performance advantage of binary search over linear search by creating an array of one million numbers and timing how long it takes the linear-Contains() and binaryContains() functions defined in this chapter to find various numbers in the array.

2. Add a counter to dfs(), bfs(), and a star() to see how many states each searches through for the same maze. Find the counts for 100 different mazes to get statistically significant results.

3. Find a solution to the missionaries and cannibals problem for a different number of starting missionaries and cannibals.
Don’t just learn another language. Become a better programmer instead. Today’s awesome iOS apps stand on the shoulders of classic algorithms, coding techniques, and engineering principles. Master these core skills in Swift, and you’ll be ready for AI, data-centric programming, machine learning, and the other development challenges that will define the next decade.

Classic Computer Science Problems in Swift deepens your Swift language skills by exploring foundational coding techniques and algorithms. As you work through examples in search, clustering, graphs, and more, you’ll remember important things you’ve forgotten and discover classic solutions to your “new” problems. You’ll appreciate author David Kopec’s amazing ability to connect the core disciplines of computer science to the real-world concerns of apps, data, performance, and even nailing your next job interview!

What’s Inside

- Breadth-first, depth-first, and A* search algorithms
- Constraint-satisfaction problems
- Solving problems with graph algorithms
- Neural networks, genetic algorithms, and more
- All examples written in Swift 4.1

For readers comfortable with the basics of Swift.

David Kopec is an assistant professor of computer science and innovation at Champlain College in Burlington, Vermont. He is an experienced iOS developer and the author of Dart for Absolute Beginners.

“A hands-on and informative exploration of computer science problems.”
—Chad Johnston, Polaris Alpha

“A fun read to sharpen your classic programming skills and to bring your Swift programming to another level.”
—Becky Huett, Big Shovel Labs

“An excellent book for all Swift programmers, and for students learning algorithms.”
—Julien Pohie, Dell

“An extraordinary Swift language book and a contribution to the practice of Swift algorithmic problem solving.”
—Eric Giannini, Nexmo

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/classic-computer-science-problems-in-swift

$39.99 / Can $52.99 [INCLUDING eBook]