Covers Swift 4.1

Classic Computer Science Problems in Swift

Essential techniques for practicing programmers

David Kopec

SAMPLE CHAPTER

MANNING

Classic Computer Science Problems in Swift

by David Kopec

Chapter 4

brief contents

4 Graph problems

A *graph* is an abstract mathematical construct that is used for modeling a real-world problem by dividing the problem into a set of connected nodes. We call each of the nodes a *vertex* and each of the connections an *edge*. For instance, a subway map can be thought of as a graph representing a transportation network. Each of the dots represents a station, and each of the lines represents a route between two stations. In graph terminology, we would call the stations "vertices" and the routes "edges."

Why is this useful? Not only do graphs help us abstractly think about a problem, they also let us apply several well-understood and performant search and optimization techniques. For instance, in the subway example, suppose we want to know the shortest route from one station to another. Or, suppose we wanted to know the minimum amount of track needed to connect all of the stations. Graph algorithms that you will learn in this chapter can solve both of those problems. Further, graph algorithms can be applied to any kind of network problem—not just transportation networks. Think of computer networks, distribution networks, and utility networks. Search and optimization problems across all of these spaces can be solved using graph algorithms.

In this chapter, we won't work with a graph of subway stations, but instead cities of the United States and potential routes between them. Figure 4.1 is a map of the continental United States and the fifteen largest metropolitan statistical areas (MSAs) in the country, as estimated by the U.S. Census Bureau.[1]

Famous entrepreneur Elon Musk has suggested building a new high-speed transportation network composed of capsules traveling in pressurized tubes. According to Musk, the capsules would travel at 700 miles per hour and be suitable

[1] Data from the United States Census Bureau's American Fact Finder, https://factfinder.census.gov/.

Figure 4.1 A map of the 15 largest MSAs in the United States

for cost-effective transportation between cities less than 900 miles apart.[2] He calls this new transportation system the "Hyperloop." In this chapter we will explore classic graph problems in the context of building out this transportation network.

Musk initially proposed the Hyperloop idea for connecting Los Angeles and San Francisco. If one were to build a national Hyperloop network, it would make sense to do so between America's largest metropolitan areas. In figure 4.2 the state outlines from figure 4.1 are removed. In addition, each of the MSAs is connected with some of its neighbors (not always its nearest neighbors, to make the graph a little more interesting).

Figure 4.2 is a graph with vertices representing the 15 largest MSAs in the United States and edges representing potential Hyperloop routes between cities. The routes were chosen for illustrative purposes. Certainly other potential routes could be part of a new Hyperloop network.

This abstract representation of a real-world problem highlights the power of graphs. Now that we have an abstraction to work with, we can ignore the geography of the United States and concentrate on thinking about the potential Hyperloop network simply in the context of connecting cities. In fact, as long as we keep the edges the same, we can think about the problem with a different looking graph. In figure 4.3, the location of Miami has moved. The graph in figure 4.3, being an abstract representation, can still address the same fundamental computational problems as the graph in figure 4.2, even if Miami is not where we would expect it. But for our sanity, we will stick with the representation in figure 4.2.

---

[2] Elon Musk, "Hyperloop Alpha," http://mng.bz/chmu.

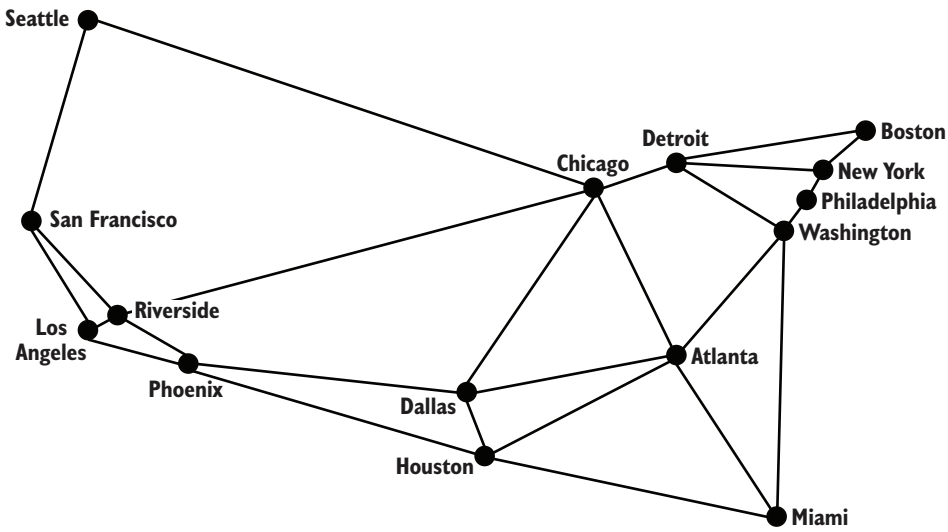

Figure 4.2 A graph with the vertices representing the 15 largest MSAs in the United States and the edges representing potential Hyperloop routes between them

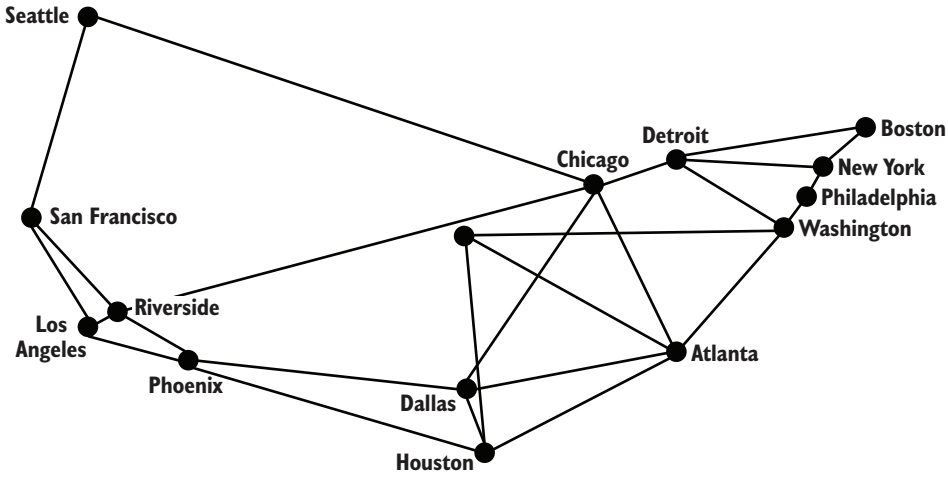

Figure 4.3 An equivalent graph to that in figure 4.2, with the location of Miami moved

4.1 Building a graph framework

Swift has been promoted as enabling a protocol-oriented style of programming (as opposed to the traditional object-oriented or functional paradigms).[3] Although the orthodoxy of this new paradigm is still being fleshed-out, what is clear is that it puts interfaces and composition ahead of inheritance. Whereas the class is the fundamental

[3] Dave Abrahams, "Protocol-Oriented Programming in Swift," WWDC 2015, Session 408, Apple Inc., http://mng.bz/zWP3.

building block in the object-oriented paradigm, and the function is the fundamental building block in functional programming, the protocol is the fundamental building block in protocol-oriented programming. In that light, we will try building a graph framework in a protocol-first style.

> **NOTE** The framework described in this section, and the examples that follow it, are largely based on a simplified version of my SwiftGraph open source project (https://github.com/davecom/SwiftGraph). SwiftGraph includes several features that go beyond the scope of this book.

We want this graph framework to be as flexible as possible, so that it can represent as many different problems as possible. To achieve this goal, we will use generics to abstract away the type of the vertices, and we will define an easy-to-adopt protocol for edges. Every vertex will ultimately be assigned an integer index, but it will be stored as the user-defined generic type.

Let's start work on the framework by defining the `Edge` protocol.

```
public protocol Edge: CustomStringConvertible {
    var u: Int { get set } // index of the "from" vertex
    var v: Int { get set } // index of the "to" vertex
    var reversed: Edge { get }
}
```

An `Edge` is defined as a connection between two vertices, each of which is represented by an integer index. By convention, `u` is used to refer to the first vertex, and `v` is used to represent the second vertex. You can also think of `u` as "from" and `v` as "to." In this chapter, we are only working with bidirectional edges (edges that can be travelled in both directions), but in *directed graphs*, also known as *digraphs*, edges can also be one-way, and the `reversed` property is meant to return an `Edge` that travels in the opposite direction. All `Edge` adoptees must implement `CustomStringConvertible` so they can be easily printed to the console.

The `Graph` protocol is about the essential role of a graph: associating vertices with edges. Again, we want to let the actual types of the vertices and edges be whatever the user of the framework desires. This lets the framework be used for a wide range of problems without needing to make intermediate data structures that glue everything together. In this light, we will use the Swift keyword `associatedtype` to define types that adopters of `Graph` can configure. For example, in a graph like the one for Hyperloop routes, we might define `VertexType` to be `String`, because we would use strings like "New York" and "Los Angeles" as the vertices. The only requirement of a potential `VertexType` is that it implements `Equatable`. `String` implements `Equatable`, so it is a valid `VertexType`.

```
protocol Graph: class, CustomStringConvertible {
    associatedtype VertexType: Equatable
    associatedtype EdgeType: Edge
    var vertices: [VertexType] { get set }
    var edges: [[EdgeType]] { get set }
}
```

The `vertices` array can be an array of any type that adopts `Equatable`. Each vertex will be stored in the array, but we will later refer to them by their integer index in the array. The vertex itself may be a complex data type, but its index will always be an `Int`, which is easy to work with. On another level, by putting this index between graph algorithms and the `vertices` array, it allows us to have two vertices that are equal in the same graph (imagine a graph with a country's cities as vertices, where the country has more than one city named "Springfield"). Even though they are the same, they will have different integer indexes.

There are many ways to implement a graph data structure, but the two most common are to use a *vertex matrix* or *adjacency lists*. In a vertex matrix, each cell of the matrix represents the intersection of two vertices in the graph, and the value of that cell indicates the connection (or lack thereof) between them. Our graph data structure uses adjacency lists. In this graph representation, every vertex has an array (or list) of vertices that it is connected to. Our specific representation uses an array of arrays of edges, so for every vertex there is an array of edges via which the vertex is connected to other vertices. `edges` is this two-dimensional array.

Notice, as well, that anything that adopts `Graph` must also adopt `class` and `CustomStringConvertible`. We want graph data structures to be reference types for memory-management purposes. It will also be slightly easier to write some of the protocol extensions if we know the adopters will be classes. `class` ensures that all graphs adopters are classes. `CustomStringConvertible` forces adopters of the protocol to be printable.

Introduced in Swift 2, protocol extensions allow fully fleshed out functions to be a part of a protocol. Amazingly, this will allow us to implement most of the functionality a graph needs before we actually define a concrete adopter of `Graph`. The following code shows the entirety of the protocol extension that adds this basic functionality, with in-source comments describing each of the functions.

```
extension Graph {
    /// How many vertices are in the graph?
    public var vertexCount: Int { return vertices.count }

    /// How many edges are in the graph?
    public var edgeCount: Int { return edges.joined().count }

    /// Get a vertex by its index.
    ///
    /// - parameter index: The index of the vertex.
    /// - returns: The vertex at i.
    public func vertexAtIndex(_ index: Int) -> VertexType {
        return vertices[index]
    }

    /// Find the first occurrence of a vertex if it exists.
    ///
    /// - parameter vertex: The vertex you are looking for.
    /// - returns: The index of the vertex. Return nil if it can't find it.
    public func indexOfVertex(_ vertex: VertexType) -> Int? {
```

```
    if let i = vertices.index(of: vertex) {
        return i
    }
    return nil
}

/// Find all of the neighbors of a vertex at a given index.
///
/// - parameter index: The index for the vertex to find the neighbors of.
/// - returns: An array of the neighbor vertices.
public func neighborsForIndex(_ index: Int) -> [VertexType] {
    return edges[index].map({self.vertices[$0.v]})
}

/// Find all of the neighbors of a given Vertex.
///
/// - parameter vertex: The vertex to find the neighbors of.
/// - returns: An optional array of the neighbor vertices.
public func neighborsForVertex(_ vertex: VertexType) -> [VertexType]? {
    if let i = indexOfVertex(vertex) {
        return neighborsForIndex(i)
    }
    return nil
}

/// Find all of the edges of a vertex at a given index.
///
/// - parameter index: The index for the vertex to find the children of.
public func edgesForIndex(_ index: Int) -> [EdgeType] {
    return edges[index]
}

/// Find all of the edges of a given vertex.
///
/// - parameter vertex: The vertex to find the edges of.
public func edgesForVertex(_ vertex: VertexType) -> [EdgeType]? {
    if let i = indexOfVertex(vertex) {
        return edgesForIndex(i)
    }
    return nil
}

/// Add a vertex to the graph.
///
/// - parameter v: The vertex to be added.
/// - returns: The index where the vertex was added.
public func addVertex(_ v: VertexType) -> Int {
    vertices.append(v)
    edges.append([EdgeType]())
    return vertices.count - 1
}

/// Add an edge to the graph.
```

```
    ///
    /// - parameter e: The edge to add.
    public func addEdge(_ e: EdgeType) {
        edges[e.u].append(e)
        edges[e.v].append(e.reversed as! EdgeType)
    }
}
```

Let's step back for a moment and consider why this protocol has two versions of most of its functions. We know from the protocol definition that the array `vertices` is an array of elements of type `VertexType`, which can be anything that implements `Equatable`. So we have vertices of type `VertexType` that are stored in the `vertices` array. But if we want to retrieve or manipulate them later, we need to know where they are stored in that array. Hence, every vertex has an index in the array (an integer) associated with it. If we don't know a vertex's index, we need to look it up by searching through `vertices`. That is why there are two versions of every function. One operates on `Int` indexes, and one operates on `VertexType` itself. The functions that operate on `VertexType` look up the relevant indices and call the index-based function.

Most of the functions are fairly self-explanatory, but `neighborsForIndex()` deserves a little unpacking. It returns the *neighbors* of a vertex. A vertex's neighbors are all of the other vertices that are directly connected to it by an edge. For example, in figure 4.2, New York and Washington are neighbors (the only neighbors) of Philadelphia. We find the neighbors for a vertex by looking at the ends (the `v`s) of all of the edges going out from it.

```
public func neighborsForIndex(_ index: Int) -> [VertexType] {
    return edges[index].map({self.vertices[$0.v]})
}
```

`edges[index]` is the adjacency list, the list of edges through which the vertex in question is connected to other vertices. In the closure of the map call, `$0` represents one particular edge, and `$0.v` represents the neighbor that the edge is connected to. `map()` will return all of the vertices (as opposed to just their indices), because `$0.v` is passed as an index into the `vertices` array.

Another important thing to note is the way `addEdge()` works. `addEdge()` first adds an edge to the adjacency list of the "from" vertex (`u`), and then adds a reversed version of itself to the adjacency list of the "to" vertex (`v`). The second step is necessary because this graph is not directed. We want every edge added to be bidirectional—that means that `u` will be a neighbor of `v` in the same way that `v` is a neighbor of `u`.

```
public func addEdge(_ e: EdgeType) {
    edges[e.u].append(e)
    edges[e.v].append(e.reversed as! EdgeType)
}
```

4.1.1 A concrete implementation of Edge

As was mentioned earlier, we are only dealing with bidirectional edges in this chapter. Beyond being bidirectional or unidirectional, edges can also be *unweighted* or *weighted*. A weighted edge is one that has some comparable value (usually numeric, but not always) associated with it. We could think of the weights in our potential Hyperloop network as being the distances between the stations. For now, though, we will deal with an unweighted version of the graph. An unweighted edge is simply a connection between two vertices. Another way of putting it is that in an unweighted graph we know which vertices are connected, whereas in a weighted graph we know which vertices are connected and we know something about those connections.

Our implementation of an unweighted edge, `UnweightedEdge`, will of course implement the `Edge` protocol. It must have a place for a "from" vertex (`u`), a place for a "to" vertex (`v`), and a way to reverse itself. It also must implement `CustomStringConvertible`, as required by `Edge`, which means having a `description` property.

```
open class UnweightedEdge: Edge {
    public var u: Int // "from" vertex
    public var v: Int // "to" vertex
    public var reversed: Edge {
        return UnweightedEdge(u: v, v: u)
    }

    public init(u: Int, v: Int) {
        self.u = u
        self.v = v
    }

    //MARK: CustomStringConvertable
    public var description: String {
        return "\(u) <-> \(v)"
    }
}
```

4.1.2 A concrete implementation of Graph

`UnweightedEdge` is pretty simple. Surprisingly, so is our concrete implementation of `Graph`. An `UnweightedGraph` is a `Graph` whose vertices can be any `Equatable` type (as per the `Graph` protocol) and whose edges are of type `UnweightedEdge`. By defining the types of the `vertices` and `edges` arrays, we are implicitly filling in the associated types `VertexType` and `EdgeType` in the `Graph` protocol.

```
open class UnweightedGraph<V: Equatable>: Graph {
    var vertices: [V] = [V]()
    var edges: [[UnweightedEdge]] = [[UnweightedEdge]]() //adjacency lists

    public init() {
    }

    public init(vertices: [V]) {
```

```
        for vertex in vertices {
            _ = self.addVertex(vertex)
        }
    }

    /// This is a convenience method that adds an unweighted edge.
    ///
    /// - parameter from: The starting vertex's index.
    /// - parameter to: The ending vertex's index.
    public func addEdge(from: Int, to: Int) {
        addEdge(UnweightedEdge(u: from, v: to))
    }

    /// This is a convenience method that adds an unweighted, undirected
        ➥ edge between the first occurrence of two vertices.
    ///
    /// - parameter from: The starting vertex.
    /// - parameter to: The ending vertex.
    public func addEdge(from: V, to: V) {
        if let u = indexOfVertex(from) {
            if let v = indexOfVertex(to) {
                addEdge(UnweightedEdge(u: u, v: v))
            }
        }
    }

    /// MARK: Implement CustomStringConvertible
    public var description: String {
        var d: String = ""
        for i in 0..<vertices.count {
            d += "\(vertices[i]) -> \(neighborsForIndex(i))\n"
        }
        return d
    }
}
```

The new abilities in `UnweightedGraph` are init methods, convenience methods for adding `UnweightedEdge`s to the graph, and the property `description` for conformance with `CustomStringConvertible`.

Now that we have concrete implementations of `Edge` and `Graph` we can actually create a representation of the potential Hyperloop network. The vertices and edges in `cityGraph` correspond to the vertices and edges represented in figure 4.2.

```
var cityGraph: UnweightedGraph<String>
➥ = UnweightedGraph<String>(vertices: ["Seattle", "San
➥ Francisco", "Los Angeles", "Riverside", "Phoenix", "Chicago",
➥ "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston",
➥ "Detroit", "Philadelphia", "Washington"])

cityGraph.addEdge(from: "Seattle", to: "Chicago")
cityGraph.addEdge(from: "Seattle", to: "San Francisco")
cityGraph.addEdge(from: "San Francisco", to: "Riverside")
cityGraph.addEdge(from: "San Francisco", to: "Los Angeles")
```

```
cityGraph.addEdge(from: "Los Angeles", to: "Riverside")
cityGraph.addEdge(from: "Los Angeles", to: "Phoenix")
cityGraph.addEdge(from: "Riverside", to: "Phoenix")
cityGraph.addEdge(from: "Riverside", to: "Chicago")
cityGraph.addEdge(from: "Phoenix", to: "Dallas")
cityGraph.addEdge(from: "Phoenix", to: "Houston")
cityGraph.addEdge(from: "Dallas", to: "Chicago")
cityGraph.addEdge(from: "Dallas", to: "Atlanta")
cityGraph.addEdge(from: "Dallas", to: "Houston")
cityGraph.addEdge(from: "Houston", to: "Atlanta")
cityGraph.addEdge(from: "Houston", to: "Miami")
cityGraph.addEdge(from: "Atlanta", to: "Chicago")
cityGraph.addEdge(from: "Atlanta", to: "Washington")
cityGraph.addEdge(from: "Atlanta", to: "Miami")
cityGraph.addEdge(from: "Miami", to: "Washington")
cityGraph.addEdge(from: "Chicago", to: "Detroit")
cityGraph.addEdge(from: "Detroit", to: "Boston")
cityGraph.addEdge(from: "Detroit", to: "Washington")
cityGraph.addEdge(from: "Detroit", to: "New York")
cityGraph.addEdge(from: "Boston", to: "New York")
cityGraph.addEdge(from: "New York", to: "Philadelphia")
cityGraph.addEdge(from: "Philadelphia", to: "Washington")
```

`cityGraph` has vertices of type `String`, and we indicate each vertex with the name of the MSA that it represents. It is irrelevant in what order we add the edges to `cityGraph`. Because we implemented `CustomStringConvertible` in `UnweightedGraph` with a nicely printed description of the graph, we can now pretty-print (that's a real term!) the graph.

```
print(cityGraph)
```

You should get output similar to the following:

```
Seattle -> ["Chicago", "San Francisco"]
San Francisco -> ["Seattle", "Riverside", "Los Angeles"]
Los Angeles -> ["San Francisco", "Riverside", "Phoenix"]
Riverside -> ["San Francisco", "Los Angeles", "Phoenix", "Chicago"]
Phoenix -> ["Los Angeles", "Riverside", "Dallas", "Houston"]
Chicago -> ["Seattle", "Riverside", "Dallas", "Atlanta", "Detroit"]
Boston -> ["Detroit", "New York"]
New York -> ["Detroit", "Boston", "Philadelphia"]
Atlanta -> ["Dallas", "Houston", "Chicago", "Washington", "Miami"]
Miami -> ["Houston", "Atlanta", "Washington"]
Dallas -> ["Phoenix", "Chicago", "Atlanta", "Houston"]
Houston -> ["Phoenix", "Dallas", "Atlanta", "Miami"]
Detroit -> ["Chicago", "Boston", "Washington", "New York"]
Philadelphia -> ["New York", "Washington"]
Washington -> ["Atlanta", "Miami", "Detroit", "Philadelphia"]
```

4.2 Finding the shortest path

The Hyperloop is so fast that, for optimizing travel time from one station to another, it probably matters less how long the distances are between the stations and more how many hops it takes (how many stations need to be visited) to get from one station to another. Each station may involve a layover, so just like with flights, the fewer stops the better.

In graph theory, a set of edges that connects two vertices is known as a *path*. In other words, a path is a way of getting from one vertex to another vertex. In the context of the Hyperloop network, a set of tubes (edges) represents the path from one city (vertex) to another (vertex). Finding optimal paths between vertices is one of the most common problems that graphs are used for.

4.2.1 Defining a path

In our graphs, a path can simply be thought of as an array of edges.

```
public typealias Path = [Edge]
```

Every `Edge` knows the index of its "from" vertex (`u`) and its "to" vertex (`v`), so given a `Graph`, it is easy to deduce the vertices that it connects. There's a method in `Graph` for that, `vertexAtIndex()`. It would be nice to have a method to pretty-print a `Path` within a `Graph`. We can do that in a short extension to `Graph`.

```
extension Graph {
    /// Prints a path in a readable format
    public func printPath(_ path: Path) {
        for edge in path {
            print("\(vertexAtIndex(edge.u)) > \(vertexAtIndex(edge.v))")
        }
    }
}
```

4.2.2 Revisiting breadth-first search (BFS)

In an unweighted graph, finding the shortest path means finding the path that has the fewest edges between the starting vertex and the destination vertex. To build out the Hyperloop network, it might make sense to first connect the furthest cities on the highly populated seaboards. That raises the question, "what is the shortest path between Boston and Miami?"

Luckily, we already know an algorithm for finding shortest paths, and we can reuse it to answer this question. Breadth-first search, introduced in chapter 2, is just as viable for graphs as it is for mazes. In fact, the mazes we worked with in chapter 2 really are graphs. The vertices are the locations in the maze, and the edges are the moves that can be made from one location to another. In an unweighted graph, a breadth-first search will find the shortest path between any two vertices.

We can rewrite the breadth-first search implementation from chapter 2 to suit working with `Graph`. We can even reuse the same `Queue` class, unchanged.

```
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 new version of `bfs()` will be an extension to `Graph`. It will no longer operate on `Node`s, as in chapter 2, but instead on vertices, referred to by their indices (`Int`s). Recall from chapter 2 that we used the `Node` class to keep track of the parent of each new `Node` we found. There was also a function, `nodeToPath()`, that used the `parent` property of each node to generate a path from the goal back to the start node (but reversed to start at the start). We will use a similar function, `pathDictToPath()`, to generate a `Path` from our starting vertex to the destination vertex.

```
/// Takes a dictionary of edges to reach each node and returns an array
    ➥ of edges
/// that goes from `from` to `to`
public func pathDictToPath(from: Int, to: Int, pathDict:
➥ [Int: Edge]) -> Path {
    if pathDict.count == 0 {
        return []
    }
    var edgePath: Path = Path()
    var e: Edge = pathDict[to]!
    edgePath.append(e)
    while (e.u != from) {
        e = pathDict[e.u]!
        edgePath.append(e)
    }
    return Array(edgePath.reversed())
}
```

In the new version of `bfs()`, in lieu of having access to the `parent` property on `Node`, we will use a dictionary associating each vertex index with the `Edge` that got us to it. This is what we will call `pathDict`. `pathDictToPath()` extrapolates from this dictionary the `Path` that connects the `from` vertex to the `to` vertex by looking at every `Edge` between `to` and `from` in `pathDict`.

As you study the implementation of `bfs()` on `Graph`, it may be helpful to flip back to the implementation of `bfs()` you are already familiar with from chapter 2. How has it changed? What has stayed the same? All of the basic machinery, aside from `pathDict`, is essentially the same, but several of the parameter types and generic types have been modified.

```
extension Graph {
    //returns a path to the goal vertex
    func bfs(initialVertex: VertexType, goalTestFn:
    ➥ (VertexType) -> Bool) -> Path? {
```

```
        guard let startIndex = indexOfVertex(initialVertex)
        ➥ else { return nil }
        // frontier is where we've yet to go
        let frontier: Queue<Int> = Queue<Int>()
        frontier.push(startIndex)
        // explored is where we've been
        var explored: Set<Int> = Set<Int>()
        explored.insert(startIndex)
        // how did we get to each vertex
        var pathDict: [Int: EdgeType] = [Int: EdgeType]()
        // keep going while there is more to explore
        while !frontier.isEmpty {
            let currentIndex = frontier.pop()
            let currentVertex = vertexAtIndex(currentIndex)
            // if we found the goal, we're done
            if goalTestFn(currentVertex) {
                return pathDictToPath(from: startIndex, to: currentIndex,
                ➥ pathDict: pathDict)
            }
            // check where we can go next and haven't explored
            for edge in edgesForIndex(currentIndex)
            ➥ where !explored.contains(edge.v) {
                explored.insert(edge.v)
                frontier.push(edge.v)
                pathDict[edge.v] = edge
            }
        }
        return nil // never found the goal
    }
}
```

The new `bfs()` takes a starting vertex, `initialVertex`, a function that will determine if the goal is reached, `goalTestFn()`, and returns an optional `Path`. The returned optional `Path` will be `nil` if `initialVertex` is not actually in the `Graph` (this is determined by the `guard` statement). It will also return `nil` if `goalTestFn()` never returns `true` for any of the searched vertices in the graph. `frontier` and `explored` are much the same as they were in chapter 2, except that now the generic type of each is set to `Int`—the index of a vertex in a `Graph`. This version of `bfs()` has no `successorFn()`. Instead, `edgesForIndex()` brings the next unexplored vertices onto the frontier. Finally, the last main difference between this version and the prior one is the use of `pathDict`, which gets updated when a new vertex is added to the queue, and which is used to return the final `Path` when the goal is found by calling `pathDictToPath()`.

We are now ready to find the shortest path (in terms of number of edges) between Boston and Miami. We can pass a closure to `bfs()` that tests for a goal of a vertex equivalent to the `String` `"Miami"`. If a `Path` is found, we can print it using the `printPath()` method introduced earlier as a protocol extension to `Graph`.

```
if let bostonToMiami = cityGraph.bfs(initialVertex: "Boston",
➥ goalTestFn: { $0 == "Miami" }) {
    cityGraph.printPath(bostonToMiami)
}
```

The output should look something like this:

```
Boston > Detroit
Detroit > Washington
Washington > Miami
```

Boston to Detroit to Washington to Miami, composed of three edges, is the shortest route between Boston and Miami in terms of number of edges. Figure 4.4 highlights this route.

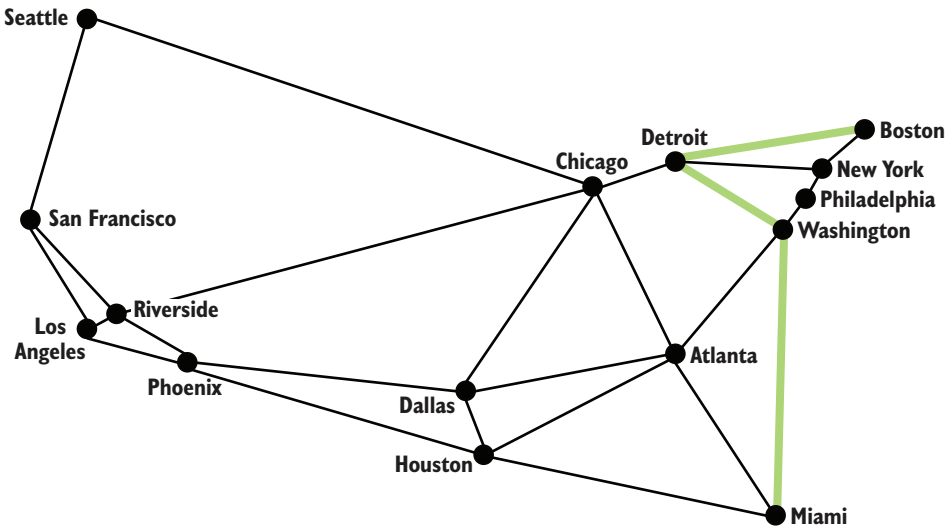

Figure 4.4 The shortest route between Boston and Miami, in terms of number of edges, is highlighted.

4.3 Minimizing the cost of building the network

Imagine we want to connect all 15 of the largest MSAs to the Hyperloop network. Our goal is to minimize the cost of rolling out the network, so that means using a minimum of track. The question is then, "how can we connect all of the MSAs using the minimum amount of track?"

4.3.1 Workings with weights

To understand the amount of track that a particular edge may require, we need to know the distance that the edge represents. This is an opportunity to re-introduce the concept of weights. In the Hyperloop network, the weight of an edge is the distance between the two MSAs that it connects. Figure 4.5 is the same as figure 4.2, except it has a weight added to each edge, representing the distance in miles between the two vertices that the edge connects.

To handle weights, we will need a new implementation of `Edge` and a new implementation of `Graph`. Once again, we want to design our framework in as flexible a way

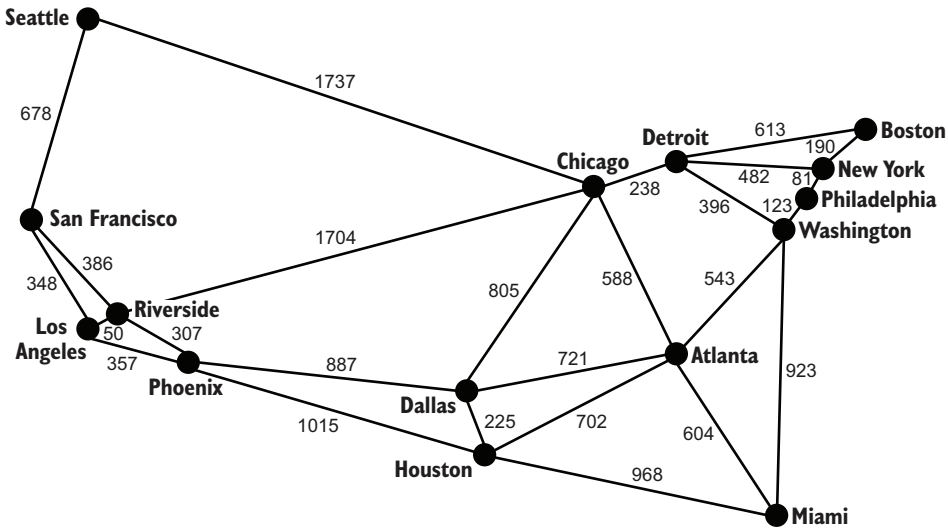

Figure 4.5 A weighted graph of the 15 largest MSAs in the United States, where each of the weights represents the distance between two MSAs in miles

as possible. To this end, we will allow the type of the weights associated with edges in our new `WeightedEdge` and `WeightedGraph` to be generic and therefore determined at creation time. But in order to execute several algorithms on weighted graphs, we do need the weights to have two properties: It must be possible to compare them, and it must be possible to add them together.

Any type that implements `Comparable` can be compared using operators like `==` and `<`. There is no built-in protocol in Swift for specifying that a type can be added, so we will create our own.

```
public protocol Summable {
    static func +(lhs: Self, rhs: Self) -> Self
}
```

If a type implements `Summable`, it means that instances of it can be added together. All of our weights must be `Summable`, meaning it must be possible to add them together, so they must implement the `+` operator. Of course, one category of types that can be added is numbers. Because the built-in number types in Swift already implement the `+` operator, it is possible to add `Summable` support to them without any work.

```
extension Int: Summable {}
extension Double: Summable {}
extension Float: Summable {}
```

A `WeightedEdge` will have a generic type, `W`, representing the type of its weight. It will also implement the protocols `Edge` and `Comparable`. Why does it implement `Comparable`?

The reason is that Jarnik's algorithm, which we will cover shortly, requires the ability to compare one edge with another.

```
open class WeightedEdge<W: Comparable & Summable>: Edge, Comparable {
    public var u: Int
    public var v: Int
    public let weight: W

    public var reversed: Edge {
        return WeightedEdge(u: v, v: u, weight: weight)
    }

    public init(u: Int, v: Int, weight: W) {
        self.weight = weight
        self.u = u
        self.v = v
    }

    //Implement CustomStringConvertible protocol
    public var description: String {
        return "\(u) <\(weight)> \(v)"
    }

    //MARK: Operator Overloads for Comparable
    static public func == <W>(lhs: WeightedEdge<W>,
    ➥ rhs: WeightedEdge<W>) -> Bool {
        return lhs.u == rhs.u && lhs.v == rhs.v && lhs.weight == rhs.weight
    }

    static public func < <W>(lhs: WeightedEdge<W>, rhs:
    ➥ WeightedEdge<W>) -> Bool {
        return lhs.weight < rhs.weight
    }
}
```

The implementation of `WeightedEdge` is not immensely different from the implementation of `UnweightedEdge`. It just has a new `weight` property and the implementation of `Comparable` via the `==` and `<` operators. The `<` operator is only interested in looking at weights, because Jarnik's algorithm is interested in finding the smallest edge by weight.

A `WeightedGraph` is a lot like an `UnweightedGraph`: It has init methods, it has convenience methods for adding `WeightedEdge`s, and it implements `CustomStringConvertible` via a `description` property. Where it differs is in the new generic type, `W`, that matches the type its weighted edges take. There is also a new method, `neighborsForIndexWithWeights()`, that returns not only each neighbor but also the weight of the edge that got to it. This method is useful for the new version of `description`.

```
open class WeightedGraph<V: Equatable & Hashable, W: Comparable & Summable>:
➥ Graph {
    var vertices: [V] = [V]()
    var edges: [[WeightedEdge<W>]] = [[WeightedEdge<W>]]() //adjacency lists
```

```
    public init() {
    }

    public init(vertices: [V]) {
        for vertex in vertices {
            _ = self.addVertex(vertex)
        }
    }

    /// Find all of the neighbors of a vertex at a given index.
    ///
    /// - parameter index: The index for the vertex to find the neighbors of.
    /// - returns: An array of tuples including the vertices as the first
        ➥ element and the weights as the second element.
    public func neighborsForIndexWithWeights(_ index: Int) -> [(V, W)] {
        var distanceTuples: [(V, W)] = [(V, W)]()
        for edge in edges[index] {
            distanceTuples += [(vertices[edge.v], edge.weight)]
        }
        return distanceTuples
    }

    /// This is a convenience method that adds a weighted edge.
    ///
    /// - parameter from: The starting vertex's index.
    /// - parameter to: The ending vertex's index.
    /// - parameter weight: the Weight of the edge to add.
    public func addEdge(from: Int, to: Int, weight:W) {
        addEdge(WeightedEdge<W>(u: from, v: to, weight: weight))
    }

    /// This is a convenience method that adds a weighted edge between the
        ➥ first occurrence of two vertices. It takes O(n) time.
    ///
    /// - parameter from: The starting vertex.
    /// - parameter to: The ending vertex.
    /// - parameter weight: the Weight of the edge to add.
    public func addEdge(from: V, to: V, weight: W) {
        if let u = indexOfVertex(from) {
            if let v = indexOfVertex(to) {
                addEdge(WeightedEdge<W>(u: u, v: v, weight:weight))
            }
        }
    }

    //Implement Printable protocol
    public var description: String {
        var d: String = ""
        for i in 0..<vertices.count {
            d += "\(vertices[i]) -> \(neighborsForIndexWithWeights(i))\n"
        }
        return d
    }
}
```

It is now possible to actually define a weighted graph. The weighted graph we will work with is a representation of figure 4.5, called `cityGraph2`.

```
let cityGraph2: WeightedGraph<String,
➥ Int> = WeightedGraph<String, Int>(vertices:
➥ ["Seattle", "San Francisco", "Los Angeles", "Riverside",
➥ "Phoenix", "Chicago", "Boston", "New York", "Atlanta",
➥ "Miami", "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"])

cityGraph2.addEdge(from: "Seattle", to: "Chicago", weight: 1737)
cityGraph2.addEdge(from: "Seattle", to: "San Francisco", weight: 678)
cityGraph2.addEdge(from: "San Francisco", to: "Riverside", weight: 386)
cityGraph2.addEdge(from: "San Francisco", to: "Los Angeles", weight: 348)
cityGraph2.addEdge(from: "Los Angeles", to: "Riverside", weight: 50)
cityGraph2.addEdge(from: "Los Angeles", to: "Phoenix", weight: 357)
cityGraph2.addEdge(from: "Riverside", to: "Phoenix", weight: 307)
cityGraph2.addEdge(from: "Riverside", to: "Chicago", weight: 1704)
cityGraph2.addEdge(from: "Phoenix", to: "Dallas", weight: 887)
cityGraph2.addEdge(from: "Phoenix", to: "Houston", weight: 1015)
cityGraph2.addEdge(from: "Dallas", to: "Chicago", weight: 805)
cityGraph2.addEdge(from: "Dallas", to: "Atlanta", weight: 721)
cityGraph2.addEdge(from: "Dallas", to: "Houston", weight: 225)
cityGraph2.addEdge(from: "Houston", to: "Atlanta", weight: 702)
cityGraph2.addEdge(from: "Houston", to: "Miami", weight: 968)
cityGraph2.addEdge(from: "Atlanta", to: "Chicago", weight: 588)
cityGraph2.addEdge(from: "Atlanta", to: "Washington", weight: 543)
cityGraph2.addEdge(from: "Atlanta", to: "Miami", weight: 604)
cityGraph2.addEdge(from: "Miami", to: "Washington", weight: 923)
cityGraph2.addEdge(from: "Chicago", to: "Detroit", weight: 238)
cityGraph2.addEdge(from: "Detroit", to: "Boston", weight: 613)
cityGraph2.addEdge(from: "Detroit", to: "Washington", weight: 396)
cityGraph2.addEdge(from: "Detroit", to: "New York", weight: 482)
cityGraph2.addEdge(from: "Boston", to: "New York", weight: 190)
cityGraph2.addEdge(from: "New York", to: "Philadelphia", weight: 81)
cityGraph2.addEdge(from: "Philadelphia", to: "Washington", weight: 123)
```

Because `WeightedGraph` implements `CustomStringConvertible`, we can print out `cityGraph2`.

```
print(cityGraph2)
```

In the output, you will see both the vertices each vertex is connected to and the weight of those connections.

```
Seattle -> [("Chicago", 1737), ("San Francisco", 678)]
San Francisco -> [("Seattle", 678), ("Riverside", 386), ("Los Angeles", 348)]
Los Angeles -> [("San Francisco", 348), ("Riverside", 50), ("Phoenix", 357)]
Riverside -> [("San Francisco", 386), ("Los Angeles", 50), ("Phoenix", 307),
➥ ("Chicago", 1704)]
Phoenix -> [("Los Angeles", 357), ("Riverside", 307), ("Dallas", 887),
➥ ("Houston", 1015)]
Chicago -> [("Seattle", 1737), ("Riverside", 1704), ("Dallas", 805),
➥ ("Atlanta", 588), ("Detroit", 238)]
Boston -> [("Detroit", 613), ("New York", 190)]
```

```
New York -> [("Detroit", 482), ("Boston", 190), ("Philadelphia", 81)]
Atlanta -> [("Dallas", 721), ("Houston", 702), ("Chicago", 588),
➥ ("Washington", 543), ("Miami", 604)]
Miami -> [("Houston", 968), ("Atlanta", 604), ("Washington", 923)]
Dallas -> [("Phoenix", 887), ("Chicago", 805), ("Atlanta", 721),
➥ ("Houston", 225)]
Houston -> [("Phoenix", 1015), ("Dallas", 225), ("Atlanta", 702),
➥ ("Miami", 968)]
Detroit -> [("Chicago", 238), ("Boston", 613), ("Washington", 396),
➥ ("New York", 482)]
Philadelphia -> [("New York", 81), ("Washington", 123)]
Washington -> [("Atlanta", 543), ("Miami", 923), ("Detroit", 396),
➥ ("Philadelphia", 123)]
```

4.3.2 Finding the minimum spanning tree

A *tree* is a special kind of graph that has one, and only one, path between any two vertices. This implies that there are no *cycles* in a tree (which is sometimes called being *acyclic*). A cycle can be thought of as a circle (in the common sense, not the geometrical sense): If it is possible to traverse a graph from a starting vertex, never repeat any edges, and get back to the same starting vertex, then it has a cycle. Any graph that is not a tree can become a tree by pruning edges. Figure 4.6 illustrates pruning an edge to turn a graph into a tree.

A *connected* graph is a graph that has some way of getting from any vertex to any other vertex (all of the graphs we are looking at in this chapter are connected). A *spanning tree* is a tree that connects every vertex in a graph. A *minimum spanning tree* is a tree that connects every vertex in a weighted graph with the minimum total weight (compared to other spanning trees). For every weighted graph, it is possible to efficiently find its minimum spanning tree.

Whew, that was a lot of terminology! The point is that finding a minimum spanning tree is the same as finding a way to connect every vertex in a weighted graph with the minimum weight. This is an important and practical problem for anyone designing a network (transportation network, computer network, and so on)—how can every node in the network be connected for the minimum cost? That cost may be in terms of wire, track, road, or anything else. For instance, for a telephone network, another way of posing the problem is, "what is the minimum length of cable one needs to connect every phone?"

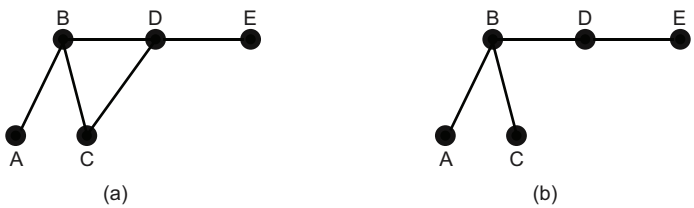

Figure 4.6 In (a), a cycle exists between vertices B, C, and D, so it is not a tree. In (b), the edge connecting C and D has been pruned, so the graph is a tree.

CALCULATING THE TOTAL WEIGHT OF A WEIGHTED PATH

Before we develop a method for finding a minimum spanning tree, we will develop a function we can use to test our future development. The solution to the minimum spanning tree problem will consist of an array of weighted edges that compose the tree. The function `totalWeight()` takes an array of `WeightedEdge<W>` and finds the total weight, `W`, that results from adding all of its edges' weights together.

```
public func totalWeight<W>(_ edges: [WeightedEdge<W>]) -> W? {
    guard let firstWeight = edges.first?.weight else { return nil }
    return edges.dropFirst().reduce(firstWeight) { (result, next) -> W in
        return result + next.weight
    }
}
```

`reduce()` is a higher-order function built in to most programming languages that can be programmed in a functional style. It takes a sequence of values and combines them via a closure. The closure is passed the result of each prior combination (the parameter `result` here) and the next value to be combined (`next` here). There's one problem—`reduce()` also requires a starting value. For most numbers, this would be 0, but because we don't know if `W` actually represents a number, we pull the first element out of `edges` and use it as the starting value. Because we do not want to re-add the first element after using it as the starting value, we call `dropFirst()` to ensure it is not added twice.

TIP `reduce()` is also known as "fold" in many other programming languages.

JARNIK'S ALGORITHM

Jarnik's algorithm for finding a minimum spanning tree works by dividing a graph into two parts: the vertices in the still-being-assembled minimum spanning tree, and the vertices not yet in the minimum spanning tree. It takes the following steps:

1. Pick an arbitrary vertex to be in the minimum spanning tree.
2. Find the lowest-weight edge connecting the minimum spanning tree to the vertices not yet in the minimum spanning tree.
3. Add the vertex at the end of that minimum edge to the minimum spanning tree.
4. Repeat steps 2 and 3 until every vertex in the graph is in the minimum spanning tree.

NOTE Jarnik's algorithm is commonly referred to as Prim's algorithm. Two Czech mathematicians, Otakar Borůvka and Vojtěch Jarník, interested in minimizing the cost of laying electric lines in the late 1920s, came up with algorithms to solve the problem of finding a minimum spanning tree. Their algorithms were "rediscovered" decades later by others.[4]

[4] Helena Durnova, "Otakar Boruvka (1899-1995) and the Minimum Spanning Tree" (Institute of Mathematics of the Czech Academy of Sciences, 2006), https://dml.cz/handle/10338.dmlcz/500001.

To run Jarnik's algorithm efficiently, a priority queue is used. Every time a new vertex is added to the minimum spanning tree, all of its outgoing edges that link to vertices outside the tree are added to the priority queue. The lowest-weight edge is always popped off the priority queue, and the algorithm keeps executing until the priority queue is empty. This ensures that the lowest-weight edges are always added to the tree first. Edges that connect to vertices already in the tree are ignored when they are popped.

The following code for `mst()` is the full implementation of Jarnik's algorithm,[5] along with a utility function for printing a `WeightedPath` and a new type defined in this extension of `WeightedGraph`.

WARNING Jarnik's algorithm will not necessarily work correctly in a graph with directed edges. It also will not work in a graph that is not connected.

```
/// Extensions to WeightedGraph for building a Minimum-Spanning Tree (MST)
public extension WeightedGraph {
    typealias WeightedPath = [WeightedEdge<W>]

    /// Find the minimum spanning tree in a weighted graph. This is the set
        ➥ of edges
    /// that touches every vertex in the graph and is of minimal combined
        ➥ weight. This function
    /// uses Jarnik's algorithm (aka Prim's algorithm) and so assumes the
        ➥ graph has
    /// undirected edges. For a graph with directed edges, the result may
        ➥ be incorrect. Also,
    /// if the graph is not fully connected, the tree will only span the
        ➥ connected component from which
    /// the starting vertex belongs.
    ///
    /// - parameter start: The index of the vertex to start creating
        ➥ the MST from.
    /// - returns: An array of WeightedEdges containing the minimum
        ➥ spanning tree, or nil if the starting vertex is invalid. If
        ➥ there are is only one vertex connected to the starting vertex,
        ➥ an empty list is returned.
    public func mst(start: Int = 0) -> WeightedPath? {
        if start > (vertexCount - 1) || start < 0 { return nil }
        var result: [WeightedEdge<W>] = [WeightedEdge<W>]() // the final
        ➥ MST goes in here
        var pq: PriorityQueue<WeightedEdge<W>> =
        ➥ PriorityQueue<WeightedEdge<W>>(ascending: true) // minPQ
        var visited: [Bool] = Array<Bool>(repeating: false, count:
        ➥ vertexCount) // already been to these

        func visit(_ index: Int) {
            visited[index] = true // mark as visited
            for edge in edgesForIndex(index) { // add all edges coming from
            ➥ here to pq
                if !visited[edge.v] { pq.push(edge) }
            }
```

[5] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition (Addison-Wesley Professional, 2011), p. 619.

```
        }

        visit(start) // the first vertex is where everything begins

        while let edge = pq.pop() { // keep going as long as there are
        ➥ edges to process
            if visited[edge.v] { continue } // if we've been both places,
            ➥ ignore
            result.append(edge) // otherwise this is the current smallest
            ➥ so add it to the result set
            visit(edge.v) // visit where this connects
        }

        return result
    }

    /// Pretty-print an edge list returned from an MST
    /// - parameter edges The edge array representing the MST
    public func printWeightedPath(_ weightedPath: WeightedPath) {
        for edge in weightedPath {
            print("\(vertexAtIndex(edge.u)) \(edge.weight)>
            ➥ \(vertexAtIndex(edge.v))")
        }
        if let tw = totalWeight(weightedPath) {
            print("Total Weight: \(tw)")
        }
    }
}
```

Let's walk through `mst()`, line by line.

```
public func mst(start: Int = 0) -> WeightedPath? {
    if start > (vertexCount - 1) || start < 0 { return nil }
```

The algorithm returns an optional `WeightedPath` representing the minimum spanning tree. It does not matter where the algorithm starts (assuming the graph is connected and undirected), so the default is set to vertex index 0. If it so happens that the `start` is invalid, `mst()` returns `nil`.

```
var result: [WeightedEdge<W>] = [WeightedEdge<W>]() // the final MST goes
➥ in here
var pq: PriorityQueue<WeightedEdge<W>> =
➥ PriorityQueue<WeightedEdge<W>>(ascending: true) // minPQ
var visited: [Bool] = Array<Bool>(repeating: false, count: vertexCount)
➥ // already been to these
```

`result` will ultimately hold the weighted path containing the minimum spanning tree. This is where we will add `WeightedEdge`s, as the lowest-weight edge is popped off and takes us to a new part of the graph. Jarnik's algorithm is considered a *greedy algorithm* because it always selects the lowest-weight edge. `pq` is where newly discovered edges are stored and the next-lowest-weight edge is popped. `visited` keeps track of

vertex indices that we have already been to. This could also have been accomplished with a `Set`, similar to `explored` in `bfs()`.

```
func visit(_ index: Int) {
    visited[index] = true // mark as visited
    for edge in edgesForIndex(index) { // add all edges coming from here
    ➥ to pq
        if !visited[edge.v] { pq.push(edge) }
    }
}
```

`visit()` is an inner convenience function that marks a vertex as visited and adds all of its edges that connect to vertices not yet visited to `pq`. Note how easy the adjacency-list model makes finding edges belonging to a particular vertex.

```
visit(start) // the first vertex is where everything begins
```

It does not matter which vertex is visited first, unless the graph is not connected. If the graph is not connected, but is instead made up of disconnected *components*, `mst()` will return a tree that spans the particular component that the starting vertex belongs to.

```
while let edge = pq.pop() { // keep going as long as there are edges to
    process
    if visited[edge.v] { continue } // if we've been both places, ignore
    result.append(edge) // otherwise this is the current smallest so add
    ➥ it to the result set
    visit(edge.v) // visit where this connects
}
return result
```

While there are still edges on the priority queue, we pop them off and check if they lead to vertices not yet in the tree. Because the priority queue is ascending, it pops the lowest-weight edges first. This ensures that the result is indeed of minimum total weight. Any edge popped that does not lead to an unexplored vertex is ignored. Otherwise, because the edge is the lowest seen so far, it is added to the result set, and the new vertex it leads to is explored. When there are no edges left to explore, the result is returned.

Let's finally return to the problem of connecting all 15 of the largest MSAs in the United States by Hyperloop, using a minimum amount of track. The route that accomplishes this is simply the minimum spanning tree of `cityGraph2`. Let's try running `mst()` on `cityGraph2`.

```
if let mst = cityGraph2.mst() {
    cityGraph2.printWeightedPath(mst)
}
```

Thanks to the pretty-printing `printWeightedPath()` method, the minimum spanning tree is easy to read.

```
Seattle 678> San Francisco
San Francisco 348> Los Angeles
Los Angeles 50> Riverside
Riverside 307> Phoenix
Phoenix 887> Dallas
Dallas 225> Houston
Houston 702> Atlanta
Atlanta 543> Washington
Washington 123> Philadelphia
Philadelphia 81> New York
New York 190> Boston
Washington 396> Detroit
Detroit 238> Chicago
Atlanta 604> Miami
Total Weight: 5372
```

In other words, this is the cumulatively shortest collection of edges that connects all of the MSAs in the weighted graph. The minimum length of track needed to connect all of them is 5372 miles. Figure 4.7 illustrates the minimum spanning tree.

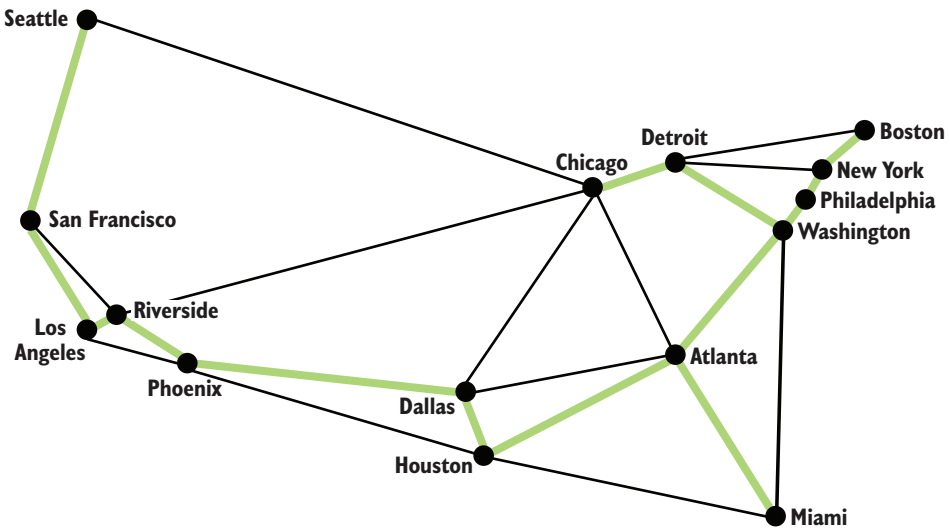

Figure 4.7 The highlighted edges represent a minimum spanning tree that connects all 15 MSAs.

4.4 Finding shortest paths in a weighted graph

As the Hyperloop network gets built, it is unlikely the builders will have the ambition to connect the whole country at once. Instead, it is likely the builders will want to minimize the cost to lay track between key cities. The cost to extend the network to particular cities will obviously depend on where the builders start.

Finding the cost to any city from some starting city is a version of the "single-source shortest path" problem. That problem asks, "what is the shortest path (in terms of total edge weight) from some vertex to every other vertex in a weighted graph?"

4.4.1 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest path problem. It is provided a starting vertex, and it returns the lowest-weight path to any other vertex on a weighted graph. It also returns the minimum total weight to every other vertex from the starting vertex. Dijkstra's algorithm starts at the single-source vertex, and then continually explores the closest vertices to the start vertex. For this reason, like Jarnik's algorithm, Dijkstra's algorithm is greedy. When Dijkstra's algorithm encounters a new vertex, it keeps track of how far it is from the start vertex, and updates this value if it ever finds a shorter path. It also keeps track of what edge got it to each vertex, like a breadth-first search.

Here are all of the algorithm's steps:

1. Add the start vertex to a priority queue.
2. Pop the closest vertex from the priority queue (at the beginning this is just the start vertex)—we'll call it the current vertex.
3. Look at all of the neighbors connected to the current vertex. If they have not previously been recorded, or the edge offers a new shortest path to them, then for each of them record its distance from the start, record the edge that produced this distance, and add the new vertex to the priority queue.
4. Repeat steps 2 and 3 until the priority queue is empty.
5. Return the shortest distance to every vertex from the start vertex and the path to get to each of them.

The extension to `WeightedGraph` for Dijkstra's algorithm includes `DijkstraNode`, a simple data structure for keeping track of costs associated with each vertex explored so far and for comparing them. This is not dissimilar to the `Node` class in chapter 2. It also includes utility functions for converting the returned array of distances to something easier to use for looking up by vertex, and for calling `dijkstra()` without vertex indices.

Without further ado, here is the code for the extension. We will go over it line by line after.

```
public extension WeightedGraph {

    /// Represents a node in the priority queue used
    /// for selecting the next
    struct DijkstraNode: Comparable, Equatable {
        let vertex: Int
        let distance: W

        public static func < (lhs: DijkstraNode, rhs: DijkstraNode) -> Bool {
            return lhs.distance < rhs.distance
        }

        public static func == (lhs: DijkstraNode, rhs: DijkstraNode)
        ➥ -> Bool {
            return lhs.distance == rhs.distance
        }
```

```
}

/// Finds the shortest paths from some route vertex to every other
    ➥ vertex in the graph.
///
/// - parameter graph: The WeightedGraph to look within.
/// - parameter root: The index of the root node to build the shortest
    ➥ paths from.
/// - parameter startDistance: The distance to get to the root node
    ➥ (typically 0).
/// - returns: Returns a tuple of two things: the first, an array
    ➥ containing the distances, the second, a dictionary containing
    ➥ the edge to reach each vertex. Use the function
    ➥ pathDictToPath() to convert the dictionary into something
    ➥ useful for a specific point.
public func dijkstra(root: Int, startDistance: W) -> ([W?],
➥ [Int: WeightedEdge<W>]) {
    var distances: [W?] = [W?](repeating: nil, count: vertexCount)
    ➥ // how far each vertex is from start
    distances[root] = startDistance // the start vertex is
    ➥ startDistance away
    var pq: PriorityQueue<DijkstraNode> =
    ➥ PriorityQueue<DijkstraNode>(ascending: true)
    var pathDict: [Int: WeightedEdge<W>] = [Int: WeightedEdge<W>]()
    ➥ // how we got to each vertex
    pq.push(DijkstraNode(vertex: root, distance: startDistance))

    while let u = pq.pop()?.vertex { // explore the next closest vertex
        guard let distU = distances[u] else { continue } // should
        ➥ already have seen it
        for we in edgesForIndex(u) { // look at every edge/vertex
        ➥ from the vertex in question
            let distV = distances[we.v] // the old distance to
            ➥ this vertex
            if distV == nil || distV! > we.weight + distU { // if
            ➥ we have no old distance or we found a shorter path
                distances[we.v] = we.weight + distU
                ➥ // update the distance to this vertex
                pathDict[we.v] = we // update the edge on the shortest
                ➥ path to this vertex
                pq.push(DijkstraNode(vertex: we.v, distance:
                ➥ we.weight + distU)) // explore it soon
            }
        }
    }

    return (distances, pathDict)
}


/// A convenience version of dijkstra() that allows the supply of
    ➥ the root
/// vertex instead of the index of the root vertex.
public func dijkstra(root: V, startDistance: W)
➥ -> ([W?], [Int: WeightedEdge<W>]) {
```

```
        if let u = indexOfVertex(root) {
            return dijkstra(root: u, startDistance: startDistance)
        }
        return ([], [:])
    }

    /// Helper function to get easier access to Dijkstra results.
    public func distanceArrayToVertexDict(distances: [W?]) -> [V : W?] {
        var distanceDict: [V: W?] = [V: W?]()
        for i in 0..<distances.count {
            distanceDict[vertexAtIndex(i)] = distances[i]
        }
        return distanceDict
    }
}
```

The first few lines of `dijkstra()` use data structures you have become familiar with, except for `distances`, which is a placeholder for the distances to every vertex in the graph from the `root`. Initially all of these distances are `nil`, because we do not yet know how far each of them is—that is what we are using Dijkstra's algorithm to figure out!

```
public func dijkstra(root: Int, startDistance: W) -> ([W?],
➥ [Int: WeightedEdge<W>]) {
    var distances: [W?] = [W?](repeating: nil, count: vertexCount)
    ➥ // how far each vertex is from start
    distances[root] = startDistance // the start vertex is startDistance away
    var pq: PriorityQueue<DijkstraNode> =
    ➥ PriorityQueue<DijkstraNode>(ascending: true)
    var pathDict: [Int: WeightedEdge<W>] = [Int: WeightedEdge<W>]()
    ➥ // how we got to each vertex
    pq.push(DijkstraNode(vertex: root, distance: startDistance))
```

The first node pushed onto the priority queue contains the root vertex.

```
while let u = pq.pop()?.vertex { // explore the next closest vertex
    guard let distU = distances[u] else { continue } // should already have
    ➥ seen it
```

We keep running Dijkstra's algorithm until the priority queue is empty. `u` is the current vertex we are searching from, and `distU` is the stored distance for getting to `u` along known routes. Every vertex explored at this stage has already been found, so it must have a known distance. If it doesn't, something is wrong, hence the `guard` statement.

```
for we in edgesForIndex(u) { // look at every edge/vertex from the vertex
➥ in question
   let distV = distances[we.v] // the old distance to this vertex
```

Next, every edge connected to `u` is explored. `distV` is the distance to any known vertex attached by an edge to `u`.

```
if distV == nil || distV! >
➥ we.weight + distU { // if we have no old distance or we found a
➥ shorter path
    distances[we.v] = we.weight + distU // update the distance to this vertex
    pathDict[we.v] = we // update the edge on the shortest path to
    ➥ this vertex
    pq.push(DijkstraNode(vertex: we.v, distance: we.weight + distU))
    ➥ // explore it soon
}
```

If we have found a vertex that has not yet been explored (`distV == nil`), or we have found a new, shorter path to it, we record that new shortest distance to `v` and the edge that got us there. It is okay to force unwrap `distV` here, because the second part of the "or" operator (`||`) is short-circuited, and we know if we get to it that `distV` is not `nil`. Finally, we push any vertices that have new paths to them to the priority queue.

```
return (distances, pathDict)
```

`dijkstra()` returns both the distances to every vertex in the weighted graph from the root vertex, and the `pathDict` that can unlock the shortest paths to them. It is safe to run Dijkstra's algorithm now. Let's start by finding the distance from Los Angeles to every other MSA in the graph.

```
let (distances, pathDict) = cityGraph2.dijkstra(root: "Los Angeles",
➥ startDistance: 0)
var nameDistance: [String: Int?] =
➥ cityGraph2.distanceArrayToVertexDict(distances: distances)
for (key, value) in nameDistance {
    print("\(key) : \(String(describing: value!))")
}
```

Your output should look something like this:

```
Phoenix : 357
Detroit : 1992
Houston : 1372
Washington : 2388
Riverside : 50
Chicago : 1754
Dallas : 1244
Atlanta : 1965
New York : 2474
Philadelphia : 2511
Boston : 2605
San Francisco : 348
Seattle : 1026
Los Angeles : 0
Miami : 2340
```

We can use our old friend, `pathDictToPath()`, to find the shortest path between Los Angeles and a specific other MSA—say Boston. Finally, we can use `printWeightedPath()` to pretty-print the result.

```
let path = pathDictToPath(from:
➥ cityGraph2.indexOfVertex("Los Angeles")!, to:
➥ cityGraph2.indexOfVertex("Boston")!, pathDict: pathDict)
cityGraph2.printWeightedPath(path as! [WeightedEdge<Int>])
```

The shortest path from Los Angeles to Boston is

```
Los Angeles 50> Riverside
Riverside 1704> Chicago
Chicago 238> Detroit
Detroit 613> Boston
Total Weight: 2605
```

You may have noticed that Dijkstra's algorithm has some resemblance to Jarnik's algorithm. They are both greedy, and it is possible to implement them using quite similar code if one is sufficiently motivated. Another algorithm that Dijkstra's algorithm resembles is A* from chapter 2. A* can be thought of as a modification of Dijkstra's algorithm. Add a heuristic and restrict Dijkstra's algorithm to finding a single destination, and the two algorithms are the same.

4.5 Real-world applications

A huge amount of our world can be represented using graphs. You have seen in this chapter how effective they are for working with transportation networks, but many other kinds of networks have the same essential optimization problems: telephone networks, computer networks, utility networks (electricity, plumbing, and so on). As a result, graph algorithms are essential for efficiency in the telecommunications, shipping, transportation, and utility industries.

Retailers must handle complex distribution problems. Stores and warehouses can be thought of as vertices and the distances between them as edges. The algorithms are the same. The internet itself is a giant graph, with each connected device a vertex and each wired or wireless connection being an edge. Whether a business is saving fuel or wire, minimum spanning tree and shortest path problem-solving are useful for more than just games. Some of the world's most famous brands became successful by optimizing graph problems: think of Walmart building out an efficient distribution network, Google indexing the web (a giant graph), and FedEx finding the right set of hubs to connect the world's addresses.

Some obvious applications of graph algorithms are social networks and map applications. In a social network, people are vertices, and connections (friendships on Facebook, for instance) are edges. In fact, one of Facebook's most prominent developer tools is known as the "Graph API" (https://developers.facebook.com/docs/graph-api). In map applications like Apple Maps and Google Maps, graph algorithms are used to provide directions and calculate trip times.

Several popular video games also make explicit use of graph algorithms. MiniMetro and Ticket to Ride are two examples of games that closely mimic the problems solved in this chapter.

4.6 Exercises

1. Add support to the graph framework for removing edges and vertices.
2. Add support to the graph framework for directed graphs (digraphs).
3. Add an extension to `Graph` for depth-first search (see chapter 2).
4. Use this chapter's graph framework to prove or disprove the classic Bridges of Konigsberg problem.

iOS/PROGRAMMING

Classic Computer Science Problems in Swift

David Kopec

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 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

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

"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

ISBN-13: 978-1-61729-489-1
ISBN-10: 1-61729-489-6
53999
9 781617 294891

MANNING $39.99 / Can $52.99 [INCLUDING eBOOK]