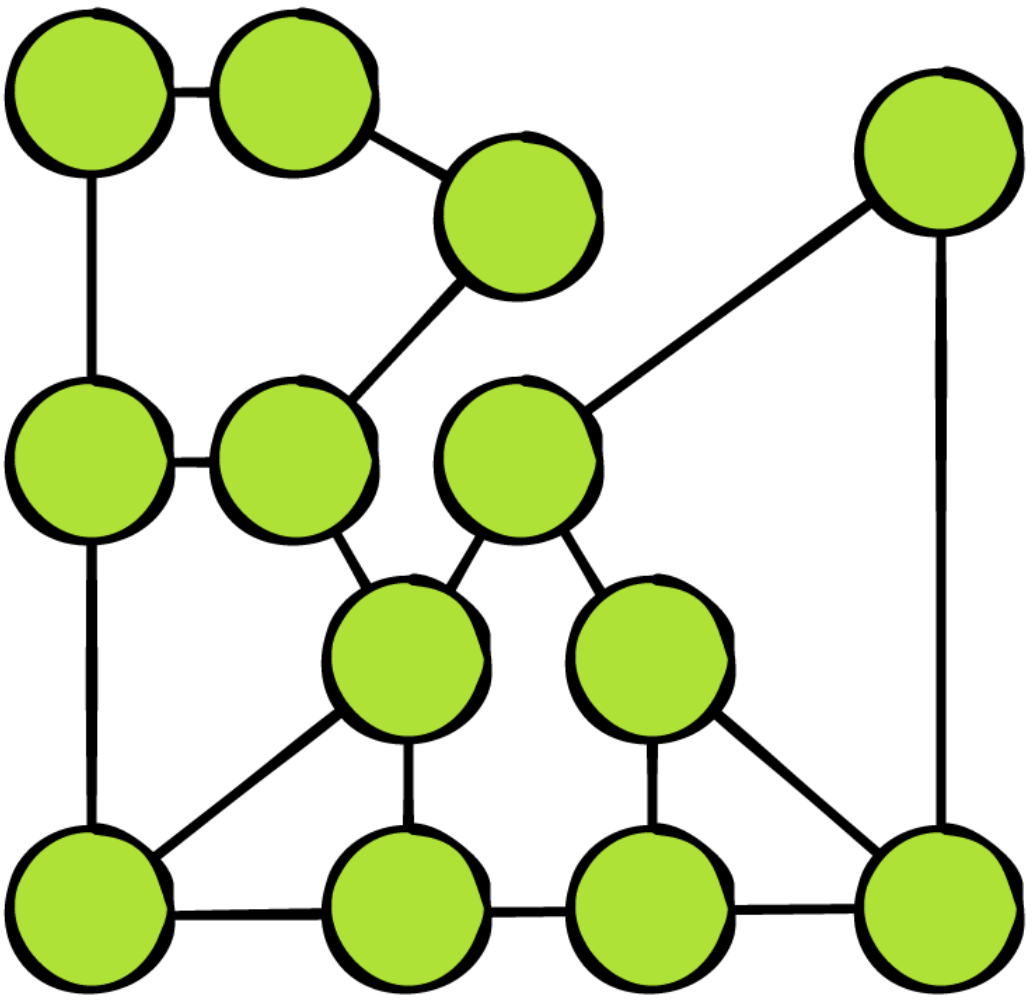
19 Graphs Written by Irina Galata

What do social networks have in common with booking cheap flights around the world? You can represent both of these real-world models as **graphs**!

A graph is a data structure that captures relationships between objects. It's made up of **vertices** connected by **edges**. In the graph below, the vertices are represented by circles, and the edges are the lines that connect them.

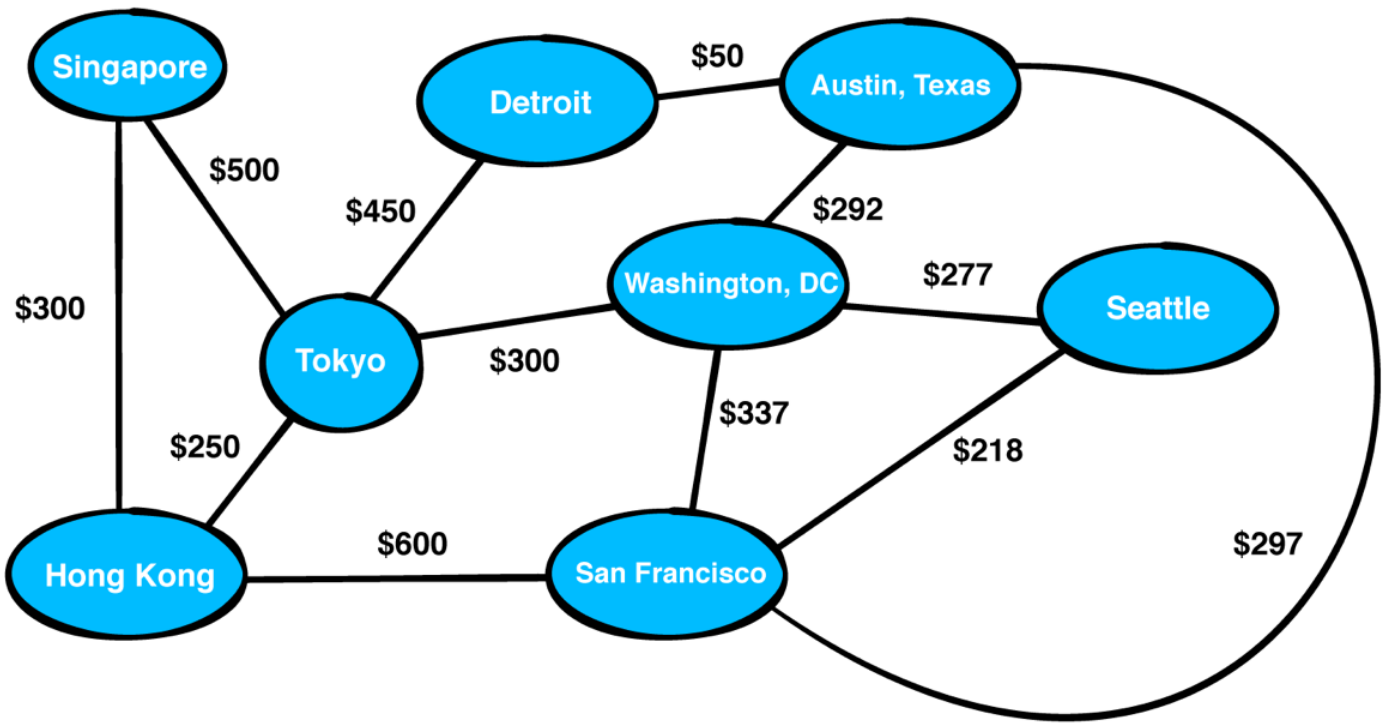


Weighted graphs

In a **weighted graph**, every edge has a weight associated with it that represents the cost of using this edge. This lets you choose the cheapest or shortest path between two vertices.

Take the airline industry as an example, and think of a network with

varying flight paths:



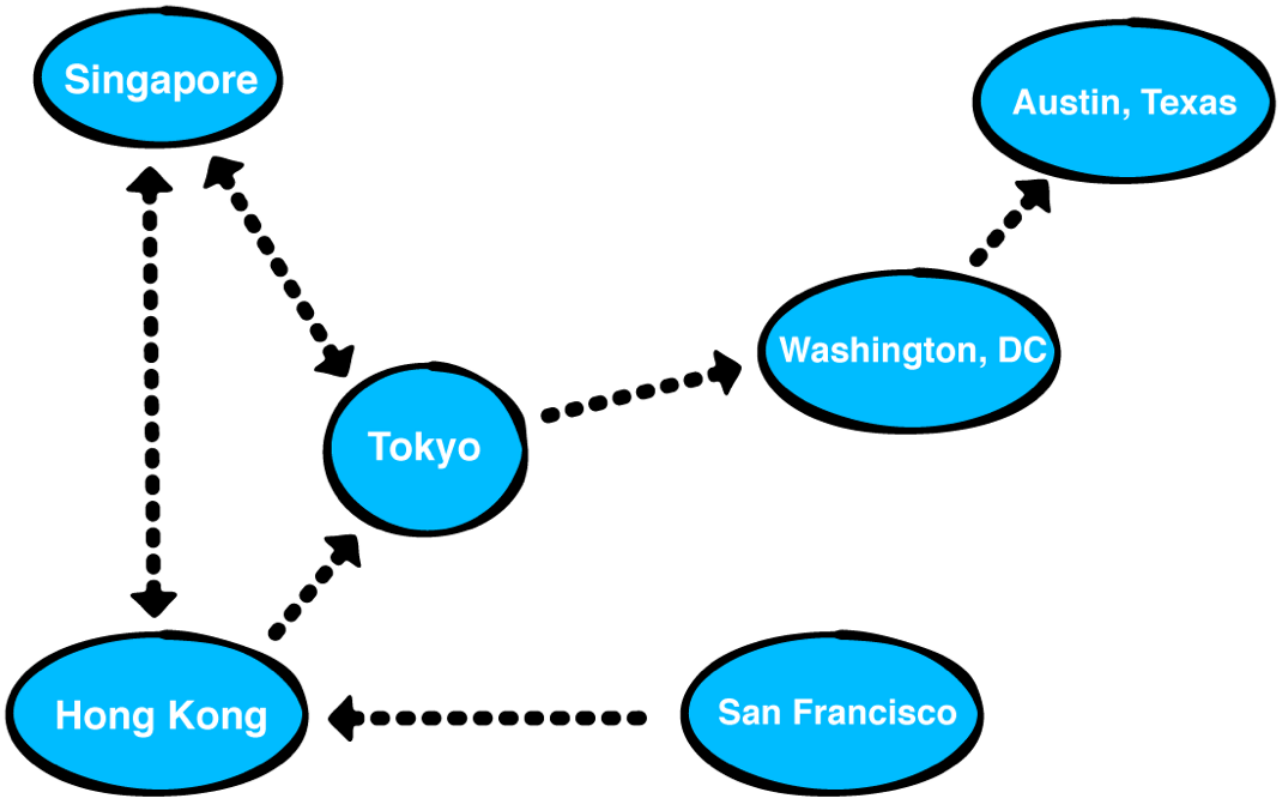
In this example, the vertices represent a state or country, while the edges represent a route from one place to another. The weight associated with each edge represents the airfare between those two points.

Using this network, you can determine the cheapest flights from San Francisco to Singapore for all those budget-minded digital nomads out there.

Directed graphs

As well as assigning a weight to an edge, your graphs can also have **direction**. Directed graphs are more restrictive to traverse, as an edge may only permit traversal in one direction.

The diagram below represents a directed graph.



A directed graph

You can tell a lot from this diagram:

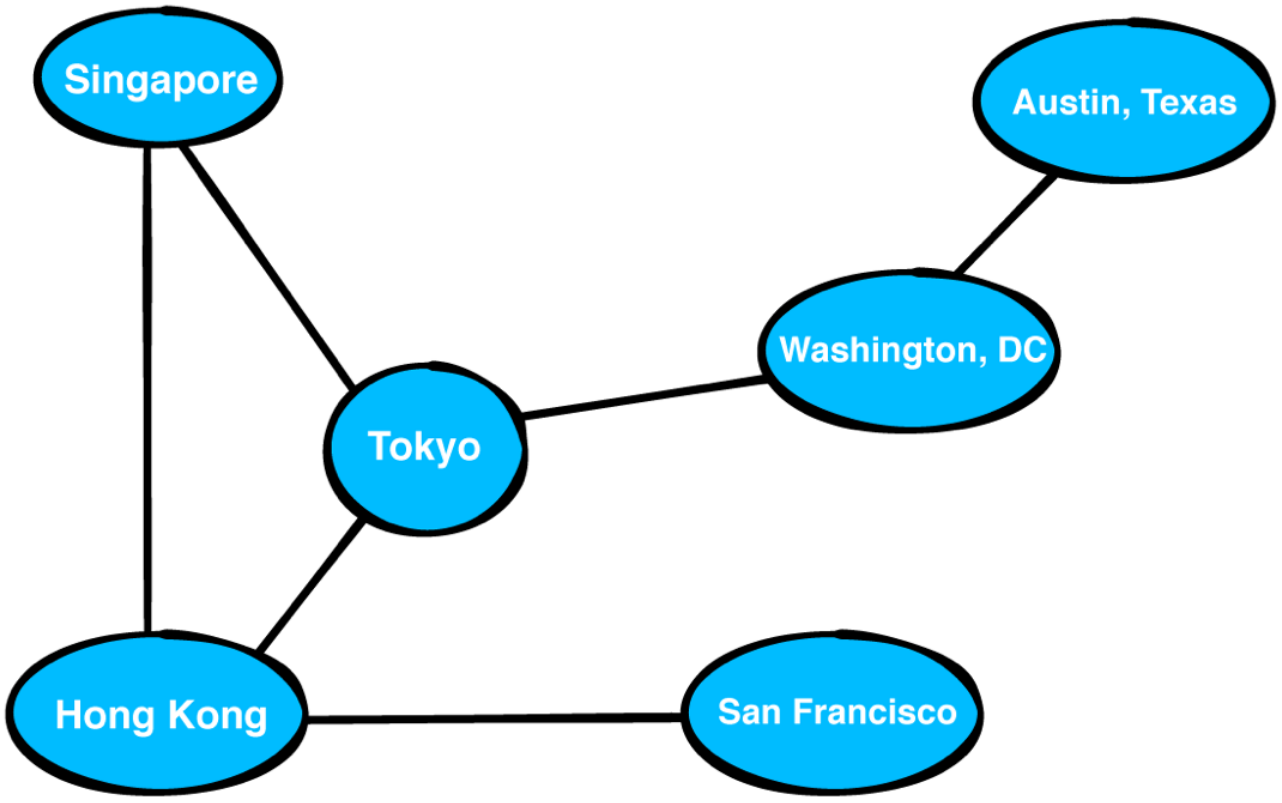
- There is a flight from Hong Kong to Tokyo.
- There is no direct flight from San Francisco to Tokyo.
- You can buy a roundtrip ticket between Singapore and Tokyo.
- There is no way to get from Tokyo to San Francisco.

Undirected graphs

You can think of an undirected graph as a directed graph where all of the edges are bi-directional.

In an undirected graph:

- Two connected vertices have edges going back and forth.
- The weight of an edge applies to both directions.



An undirected graph

Common operations

It's time to establish an interface for graphs.

Open the starter project for this chapter. Create a new file named **Graph.kt** and add the following inside the file:

```
interface Graph<T: Any> {

  fun createVertex(data: T): Vertex<T>

  fun addDirectedEdge(source: Vertex<T>,
                      destination: Vertex<T>,
                      weight: Double?)

  fun addUndirectedEdge(source: Vertex<T>,
                        destination: Vertex<T>,
                        weight: Double?)

  fun add(edge: EdgeType,
          source: Vertex<T>,
          destination: Vertex<T>,
          weight: Double?)
```

```
  fun edges(source: Vertex<T>): ArrayList<Edge<T>>

  fun weight(source: Vertex<T>,
             destination: Vertex<T>): Double?

}

enum class EdgeType {
  DIRECTED,
  UNDIRECTED
}
```

This interface describes the common operations for a graph:

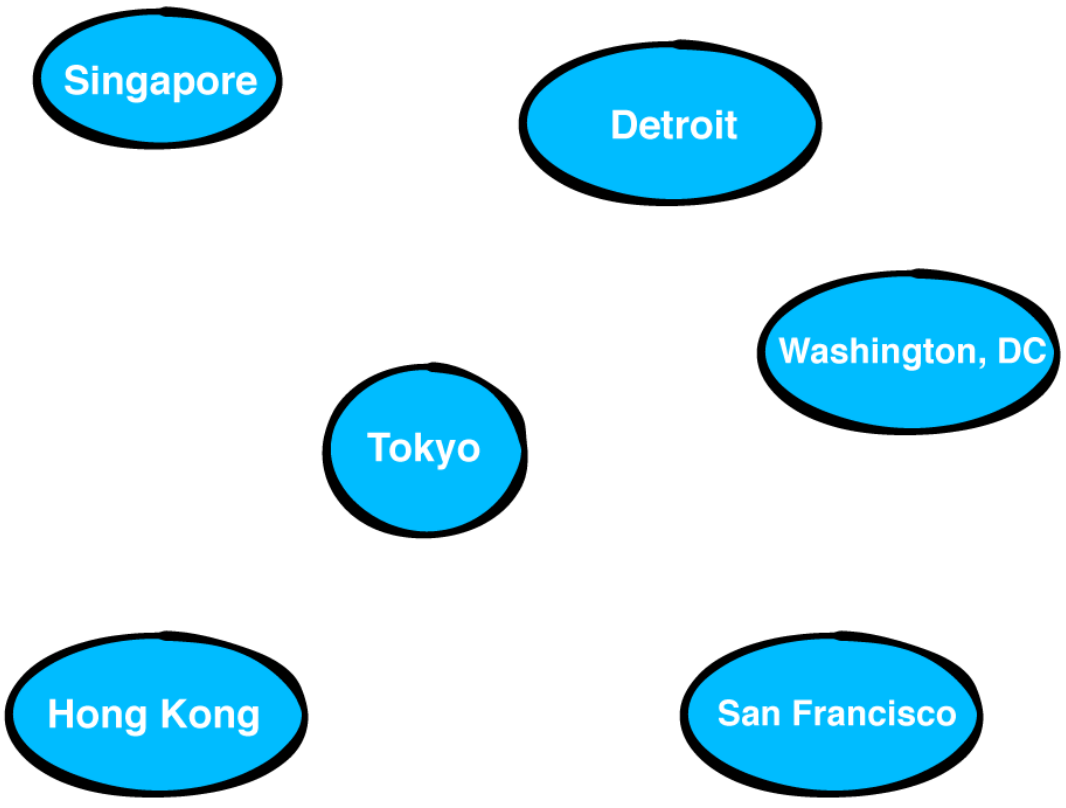
- **`createVertex()`**: Creates a vertex and adds it to the graph.
- **`addDirectedEdge()`**: Adds a directed edge between two vertices.
- **`addUndirectedEdge()`**: Adds an undirected (or bi-directional) edge between two vertices.
- **`add()`**: Uses `EdgeType` to add either a directed or undirected edge between two vertices.
- **`edges()`**: Returns a list of outgoing edges from a specific vertex.
- **`weight()`**: Returns the weight of the edge between two vertices.

In the following sections, you'll implement this interface in two ways:

- Using an adjacency list.
- Using an adjacency matrix.

Before you can do that, you must first build types to represent vertices and edges.

Defining a vertex



A collection of vertices — not yet a graph

Create a new file named **Vertex.kt** and add the following inside the file:

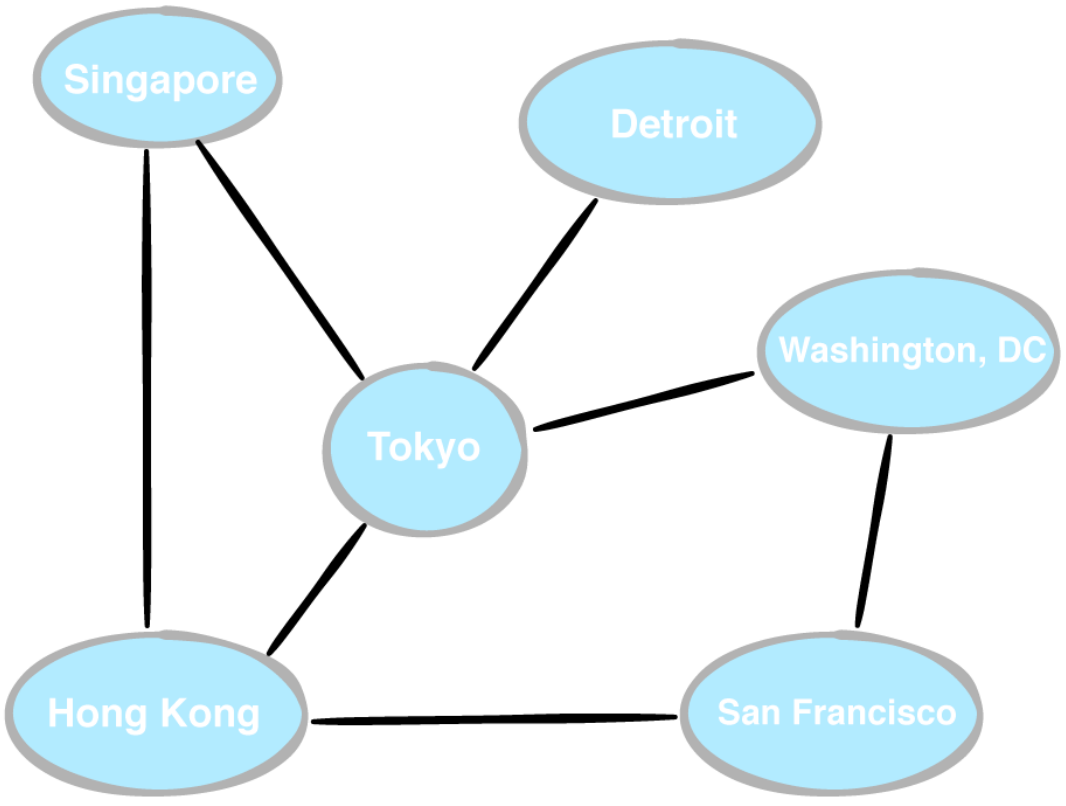
```
data class Vertex<T: Any>(val index: Int, val data: T)
```

Here, you've defined a generic `Vertex` class. A vertex has a unique index within its graph and holds a piece of data.

You defined `Vertex` as a **data class** because it will be used as a key in a `Map` later, and a data class gives you `equals()` and `hashCode()` implementations, without having to write them yourself.

Defining an edge

To connect two vertices, there must be an edge between them.



Edges added to the collection of vertices

Create a new file named **Edge.kt** and add the following inside the file:

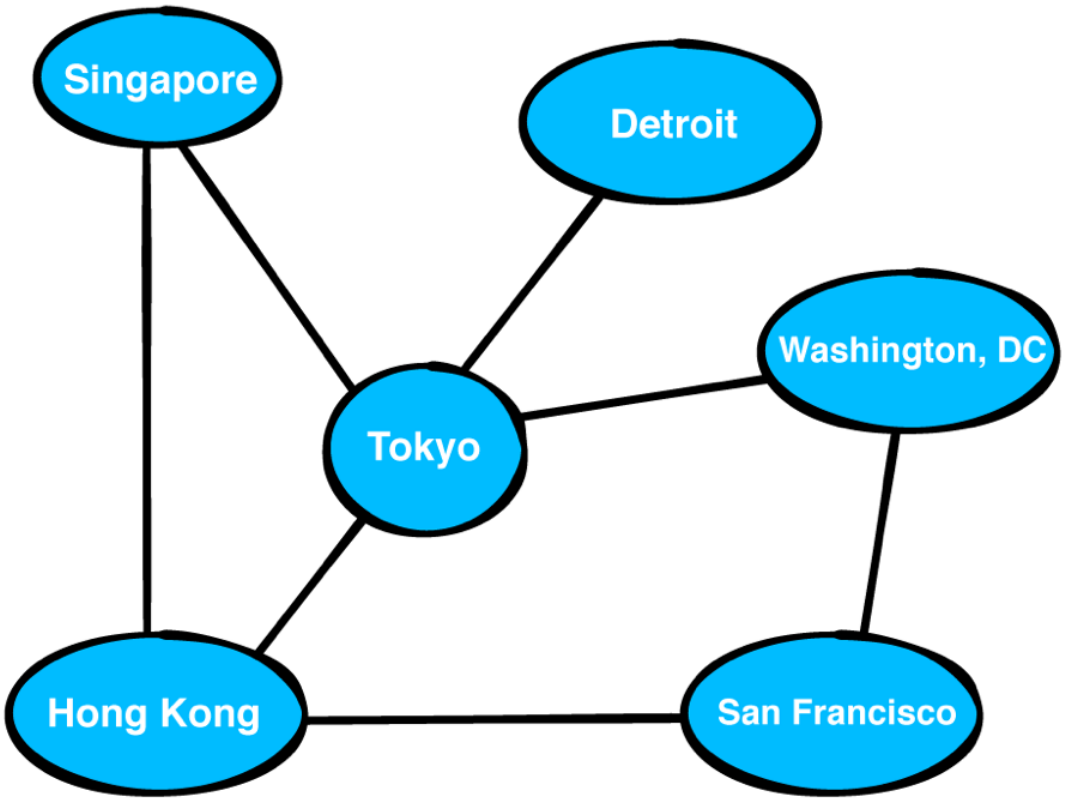
```
data class Edge<T: Any>(
  val source: Vertex<T>,
  val destination: Vertex<T>,
  val weight: Double? = null
)
```

An `Edge` connects two vertices and has an optional weight.

Adjacency list

The first graph implementation that you'll learn uses an **adjacency list**. For every vertex in the graph, the graph stores a list of outgoing edges.

Take as an example the following network:



The adjacency list below describes the network of flights depicted above:

There's a lot you can learn from this adjacency list:

1. Singapore's vertex has two outgoing edges. There's a flight from Singapore to Tokyo and Hong Kong.
2. Detroit has the smallest number of outgoing traffic.
3. Tokyo is the busiest airport, with the most outgoing flights.

In the next section, you'll create an adjacency list by storing a **map of arrays**. Each key in the map is a vertex, and in every vertex, the map holds a corresponding array of edges.

Implementation

Create a new file named **AdjacencyList.kt** and add the following:

```
class AdjacencyList<T: Any> : Graph<T> {

  private val adjacencies: HashMap<Vertex<T>, ArrayList<Edge<T>>> = HashMap

  // more to come ...
}
```

Here, you've defined an `AdjacencyList` that uses a map to store the edges.

You've already implemented the `Graph` interface but still need to implement its requirements. That's what you'll do in the following sections.

Creating a vertex

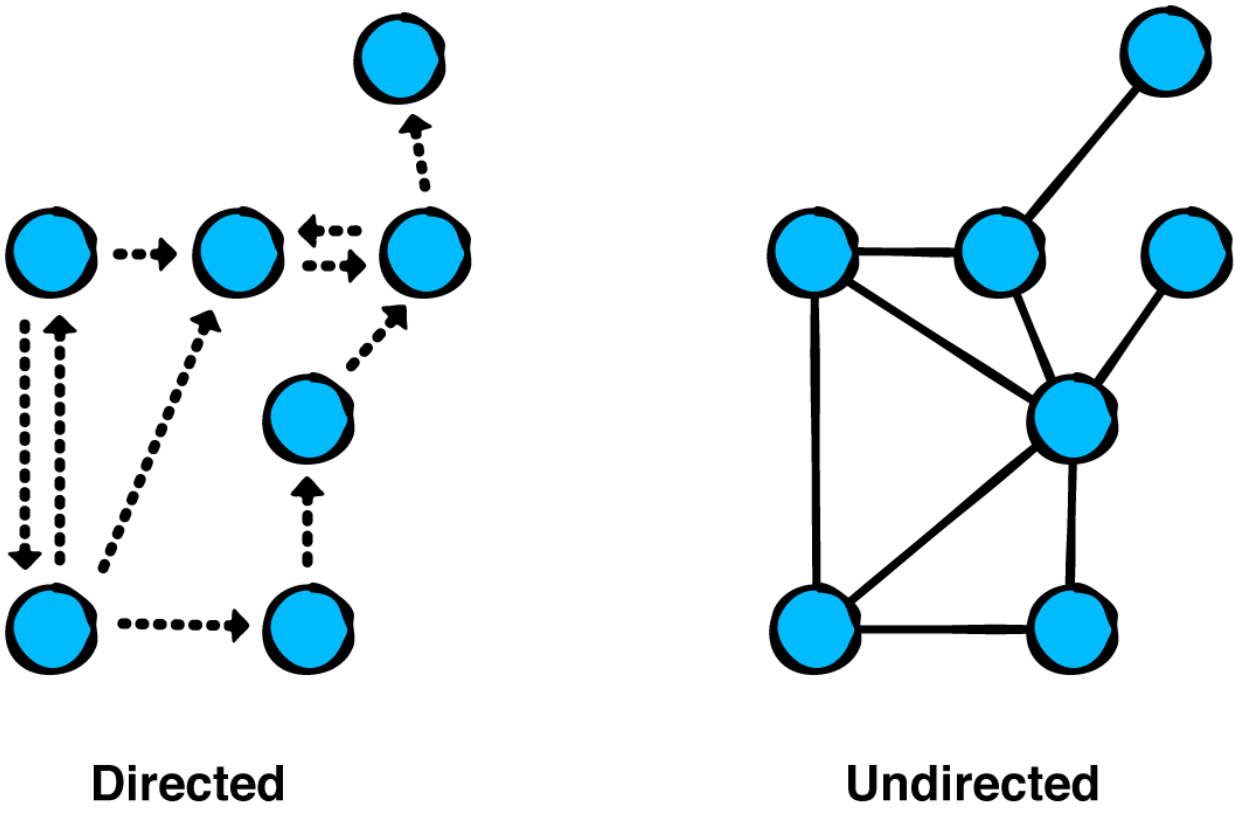
Add the following method to `AdjacencyList`:

```
override fun createVertex(data: T): Vertex<T> {
  val vertex = Vertex(adjacencies.count(), data)
  adjacencies[vertex] = ArrayList()
  return vertex
}
```

Here, you create a new vertex and return it. In the adjacency list, you store an empty list of edges for this new vertex.

Creating a directed edge

Recall that there are **directed** and **undirected** graphs.



Start by implementing the `addDirectedEdge` requirement. Add the following method:

```
override fun addDirectedEdge(source: Vertex<T>, destination: Vertex<T>, wei
  val edge = Edge(source, destination, weight)
  adjacencies[source]?.add(edge)
}
```

This method creates a new edge and stores it in the adjacency list.

Creating an undirected edge

You just created a method to add a directed edge between two vertices. How would you create an undirected edge between two vertices?

Remember that an undirected graph can be viewed as a bidirectional graph. Every edge in an undirected graph can be traversed in both directions. This is why you'll implement `addUndirectedEdge()` on top of `addDirectedEdge()`. Because this implementation is reusable, you'll add it as a default implementation in `Graph`.

In **Graph.kt**, update `addUndirectedEdge()`:

```
fun addUndirectedEdge(source: Vertex<T>, destination: Vertex<T>, weight: Do
  addDirectedEdge(source, destination, weight)
  addDirectedEdge(destination, source, weight)
}
```

Adding an undirected edge is the same as adding two directed edges.

Now that you've implemented both `addDirectedEdge()` and `addUndirectedEdge()`, you can implement `add()` by delegating to one of these methods. Update the `add()` method of `Graph` as well:

```
fun add(edge: EdgeType, source: Vertex<T>, destination: Vertex<T>, weight:
  when (edge) {
    EdgeType.DIRECTED -> addDirectedEdge(source, destination, weight)
    EdgeType.UNDIRECTED -> addUndirectedEdge(source, destination, weight)
  }
}
```

`add()` is a convenient helper method that creates either a directed or undirected edge. This is where interfaces with default methods can become very powerful. Anyone that implements the `Graph` interface only needs to implement `addDirectedEdge()` in order to get `addUndirectedEdge()` and `add()` for free.

Retrieving the outgoing edges from a vertex

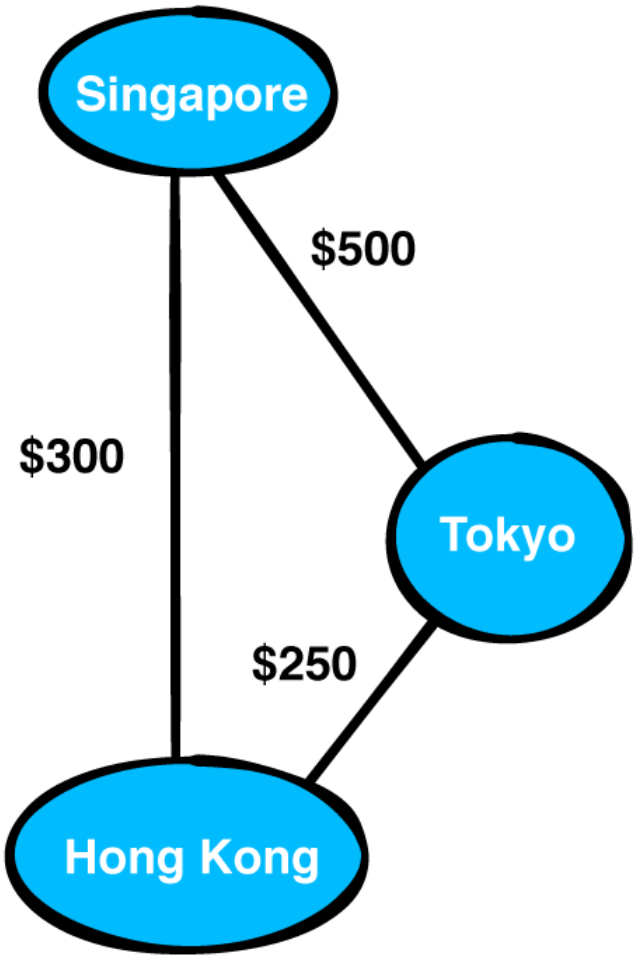
Back in **AdjacencyList.kt**, continue your work on implementing `Graph` by adding the following method:

```
override fun edges(source: Vertex<T>) =
  adjacencies[source] ?: arrayListOf()
```

This is a straightforward implementation: You either return the stored edges or an empty list if the `source` vertex is unknown.

Retrieving the weight of an edge

How much is the flight from Singapore to Tokyo?



Add the following immediately after `edges()`:

```
override fun weight(source: Vertex<T>, destination: Vertex<T>): Double? {
  return edges(source).firstOrNull { it.destination == destination }?.weigh
}
```

Here, you find the first edge from `source` to `destination`; if there is one, you return its weight.

Visualizing the adjacency list

Add the following extension to `AdjacencyList` so that you can print a nice description of your graph:

```
override fun toString(): String {
```

```
  return buildString { // 1
    adjacencies.forEach { (vertex, edges) -> // 2
      val edgeString = edges.joinToString { it.destination.data.toString()
      append("${vertex.data} ---> [ $edgeString ]\n") // 4
    }
  }
}
```

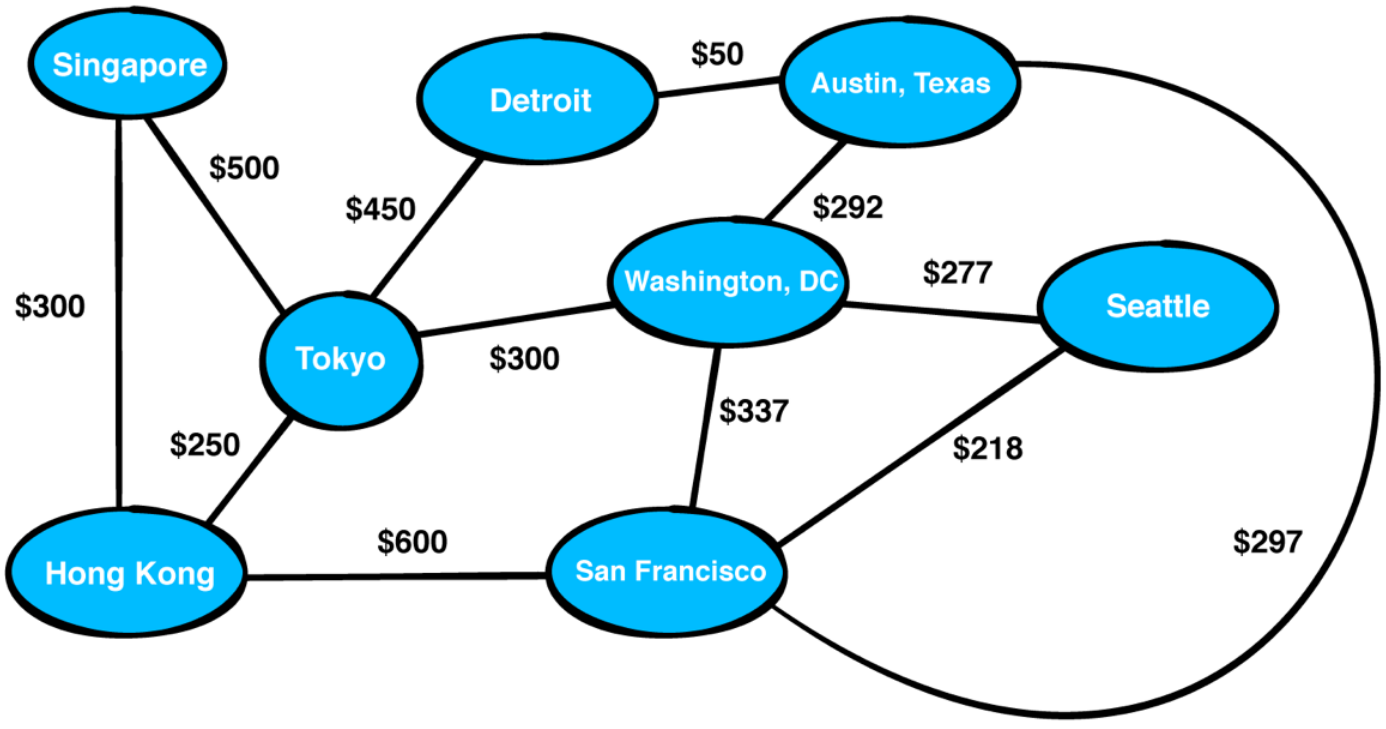
Here's what's going on in the previous code:

1. You'll be assembling the result using `buildString()`, which places you inside the scope of a `StringBuilder`, and returns whatever you've built.
2. You loop through every key-value pair in `adjacencies`.
3. For every vertex, you create a string representation of all its outgoing edges. `joinToString()` gives you a neat, comma-separated list of the items.
4. Finally, for every vertex, you append both the vertex itself and its outgoing edges to the `StringBuilder` that `buildString()` provides you with.

You've finally completed your first graph. You're ready to try it out by building a network.

Building a network

Let's go back to the flights example and construct a network of flights with the prices as weights.



Within `main()`, add the following code:

```
val graph = AdjacencyList<String>()

val singapore = graph.createVertex("Singapore")
val tokyo = graph.createVertex("Tokyo")
val hongKong = graph.createVertex("Hong Kong")
val detroit = graph.createVertex("Detroit")
val sanFrancisco = graph.createVertex("San Francisco")
val washingtonDC = graph.createVertex("Washington DC")
val austinTexas = graph.createVertex("Austin Texas")
val seattle = graph.createVertex("Seattle")

graph.add(EdgeType.UNDIRECTED, singapore, hongKong, 300.0)
graph.add(EdgeType.UNDIRECTED, singapore, tokyo, 500.0)
graph.add(EdgeType.UNDIRECTED, hongKong, tokyo, 250.0)
graph.add(EdgeType.UNDIRECTED, tokyo, detroit, 450.0)
graph.add(EdgeType.UNDIRECTED, tokyo, washingtonDC, 300.0)
graph.add(EdgeType.UNDIRECTED, hongKong, sanFrancisco, 600.0)
graph.add(EdgeType.UNDIRECTED, detroit, austinTexas, 50.0)
graph.add(EdgeType.UNDIRECTED, austinTexas, washingtonDC, 292.0)
graph.add(EdgeType.UNDIRECTED, sanFrancisco, washingtonDC, 337.0)
graph.add(EdgeType.UNDIRECTED, washingtonDC, seattle, 277.0)
graph.add(EdgeType.UNDIRECTED, sanFrancisco, seattle, 218.0)
graph.add(EdgeType.UNDIRECTED, austinTexas, sanFrancisco, 297.0)
```

```
println(graph)
```

You'll get the following output in your console:

```
Detroit ---> [ Tokyo, Austin, Texas ]
Hong Kong ---> [ Singapore, Tokyo, San Francisco ]
Singapore ---> [ Hong Kong, Tokyo ]
Washington, DC ---> [ Tokyo, Austin, Texas, San Francisco, Seattle ]
Tokyo ---> [ Singapore, Hong Kong, Detroit, Washington, DC ]
San Francisco ---> [ Hong Kong, Washington, DC, Seattle, Austin, Texas ]
Austin, Texas ---> [ Detroit, Washington, DC, San Francisco ]
Seattle ---> [ Washington, DC, San Francisco ]
```

Pretty cool, huh? This shows a visual description of an adjacency list. You can clearly see all of the outbound flights from any place.

You can also obtain other useful information such as:

- How much is a flight from Singapore to Tokyo?

```
println(graph.weight(singapore, tokyo))
```

- What are all the outgoing flights from San Francisco?

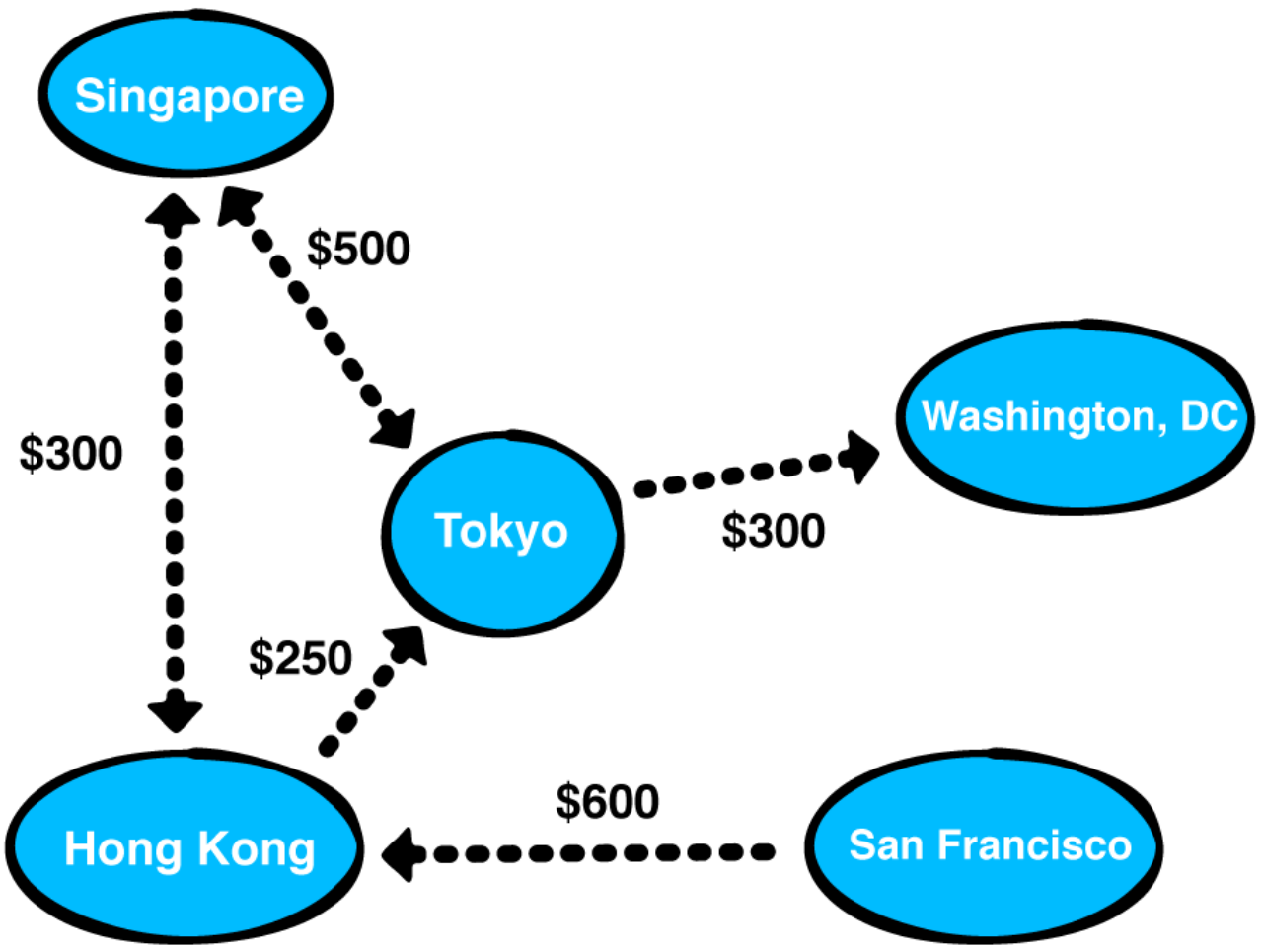
```
println("San Francisco Outgoing Flights:")
println("-------------------------------")
graph.edges(sanFrancisco).forEach { edge ->
  println("from: ${edge.source.data} to: ${edge.destination.data}")
}
```

You have just created a graph using an adjacency list, wherein you used a map to store the outgoing edges for every vertex. Let's take a look at a different approach to how to store vertices and edges.

Adjacency matrix

An **adjacency matrix** uses a square matrix to represent a graph. This matrix is a two-dimensional array wherein the value of `matrix[row][column]` is the weight of the edge between the vertices at `row` and `column`.

Below is an example of a directed graph that depicts a flight network traveling to different places. The weight represents the cost of the airfare.



The following adjacency matrix describes the network for the flights depicted above.

Edges that don't exist have a weight of 0.

Compared to an adjacency list, this matrix is a little more challenging to read.

Using the array of vertices on the left, you can learn a lot from the matrix. For example:

- `[0][1]` is 300, so there is a flight from Singapore to Hong Kong for $300.
- `[2][1]` is 0, so there is no flight from Tokyo to Hong Kong.
- `[1][2]` is 250, so there is a flight from Hong Kong to Tokyo for $250.
- `[2][2]` is 0, so there is no flight from Tokyo to Tokyo!

> **Note**: There's a pink line in the middle of the matrix. When the row and column are equal, this represents an edge between a vertex and itself, which is not allowed.

Implementation

Create a new file named **AdjacencyMatrix.kt** and add the following to it:

```
class AdjacencyMatrix<T: Any> : Graph<T> {

  private val vertices = arrayListOf<Vertex<T>>()
  private val weights = arrayListOf<ArrayList<Double?>>()
```

```
  // more to come ...
}
```

Here, you've defined an `AdjacencyMatrix` that contains an array of vertices and an adjacency matrix to keep track of the edges and their weights.

Just as before, you've already declared the implementation of `Graph` but still need to implement the requirements.

Creating a Vertex

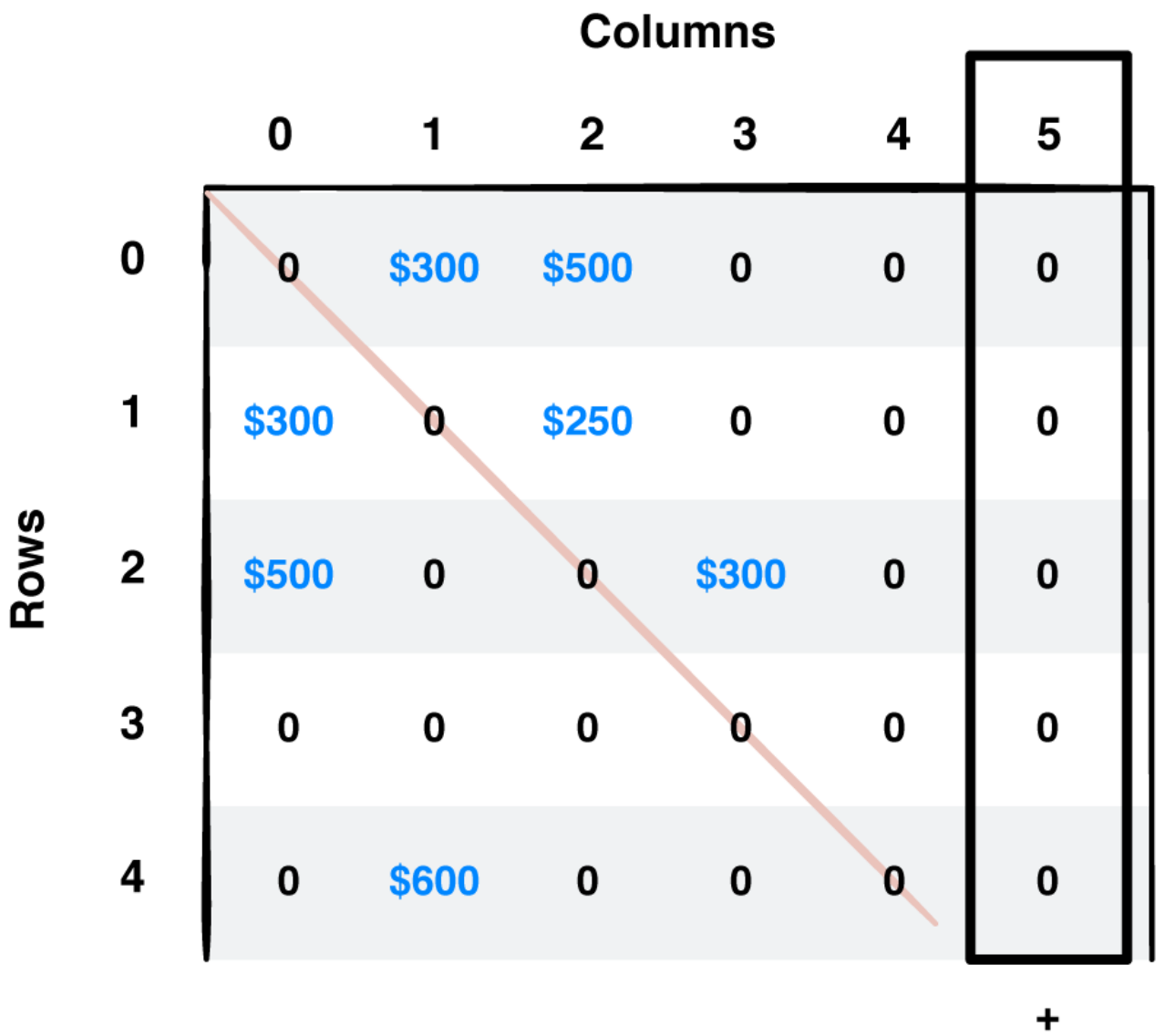
Add the following method to `AdjacencyMatrix`:

```
override fun createVertex(data: T): Vertex<T> {
  val vertex = Vertex(vertices.count(), data)
  vertices.add(vertex) // 1
  weights.forEach {
    it.add(null) // 2
  }
  weights.add(arrayListOf()) // 3
  return vertex
}
override fun createVertex(data: T): Vertex<T> {
  val vertex = Vertex(vertices.count(), data)
  vertices.add(vertex) // 1
  weights.forEach {
    it.add(null) // 2
  }
  val row = ArrayList<Double?>(vertices.count())
  repeat(vertices.count()) {
    row.add(null)
  }
  weights.add(row) // 3
  return vertex
}
```

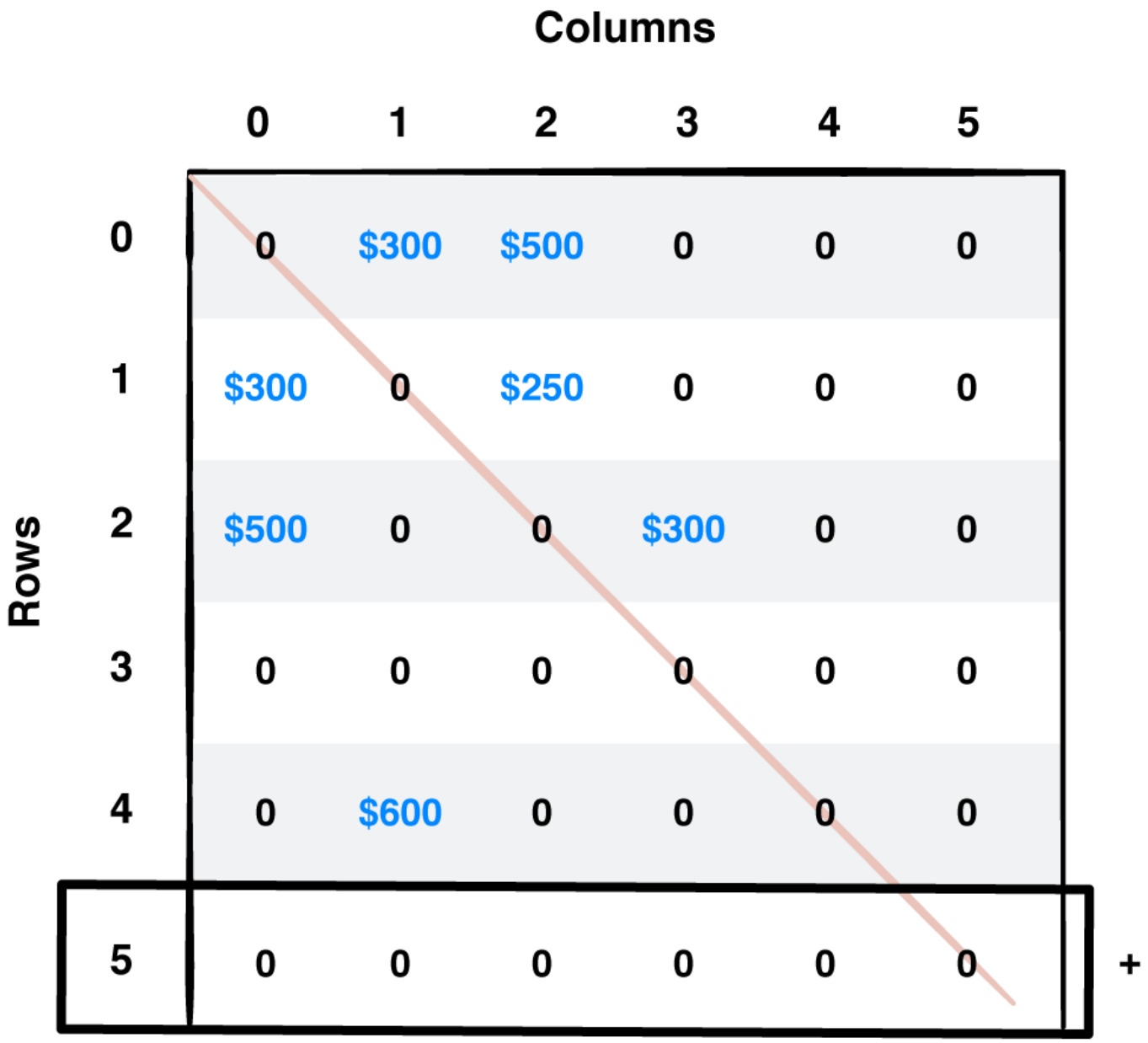
To create a vertex in an adjacency matrix, you:

1. Add a new vertex to the array.

2. Append a `null` weight to every row in the matrix, as none of the current vertices have an edge to the new vertex.



3. Add a new row to the matrix. This row holds the outgoing edges for the new vertex. You put a `null` value in this row for each vertex that your graph stores.



Creating edges

Creating edges is as simple as filling in the matrix. Add the following method:

```
override fun addDirectedEdge(
    source: Vertex<T>,
    destination: Vertex<T>,
    weight: Double?
) {
  weights[source.index][destination.index] = weight
}
```

Remember that `addUndirectedEdge()` and `add()` have a default implementation in the interface, so this is all you need to do.

Retrieving the outgoing edges from a vertex

Add the following method:

```
override fun edges(source: Vertex<T>): ArrayList<Edge<T>> {
  val edges = arrayListOf<Edge<T>>()
  (0 until weights.size).forEach { column ->
    val weight = weights[source.index][column]
    if (weight != null) {
      edges.add(Edge(source, vertices[column], weight))
    }
  }
  return edges
}
```

To retrieve the outgoing edges for a vertex, you search the row for this vertex in the matrix for weights that are not `null`.

Every non-`null` weight corresponds with an outgoing edge. The destination is the vertex that corresponds with the column in which the weight was found.

Retrieving the weight of an edge

It's easy to get the weight of an edge; simply look up the value in the adjacency matrix. Add this method:

```
override fun weight(
    source: Vertex<T>,
    destination: Vertex<T>
): Double? {
  return weights[source.index][destination.index]
}
```

Visualize an adjacency matrix

Finally, add the following extension so you can print a nice, readable description of your graph:

```
override fun toString(): String {
  // 1
  val verticesDescription = vertices.joinToString("\n") { "${it.index}: ${i

  // 2
  val grid = arrayListOf<String>()
  weights.forEach {
    var row = ""
    (0 until weights.size).forEach { columnIndex ->
      if (columnIndex >= it.size) {
        row += "ø\t\t"
      } else {
        row += it[columnIndex]?.let { "$it\t" } ?: "ø\t\t"
      }
    }
    grid.add(row)
  }
  val edgesDescription = grid.joinToString("\n")
  // 3
  return "$verticesDescription\n\n$edgesDescription"
}

override fun toString(): String {
  // 1
  val verticesDescription = vertices
      .joinToString(separator = "\n") { "${it.index}: ${it.data}" }

  // 2
  val grid = weights.map { row ->
    buildString {
      (0 until weights.size).forEach { columnIndex ->
        val value = row[columnIndex]
        if (value != null) {
          append("$value\t")
        } else {
          append("ø\t\t")
```

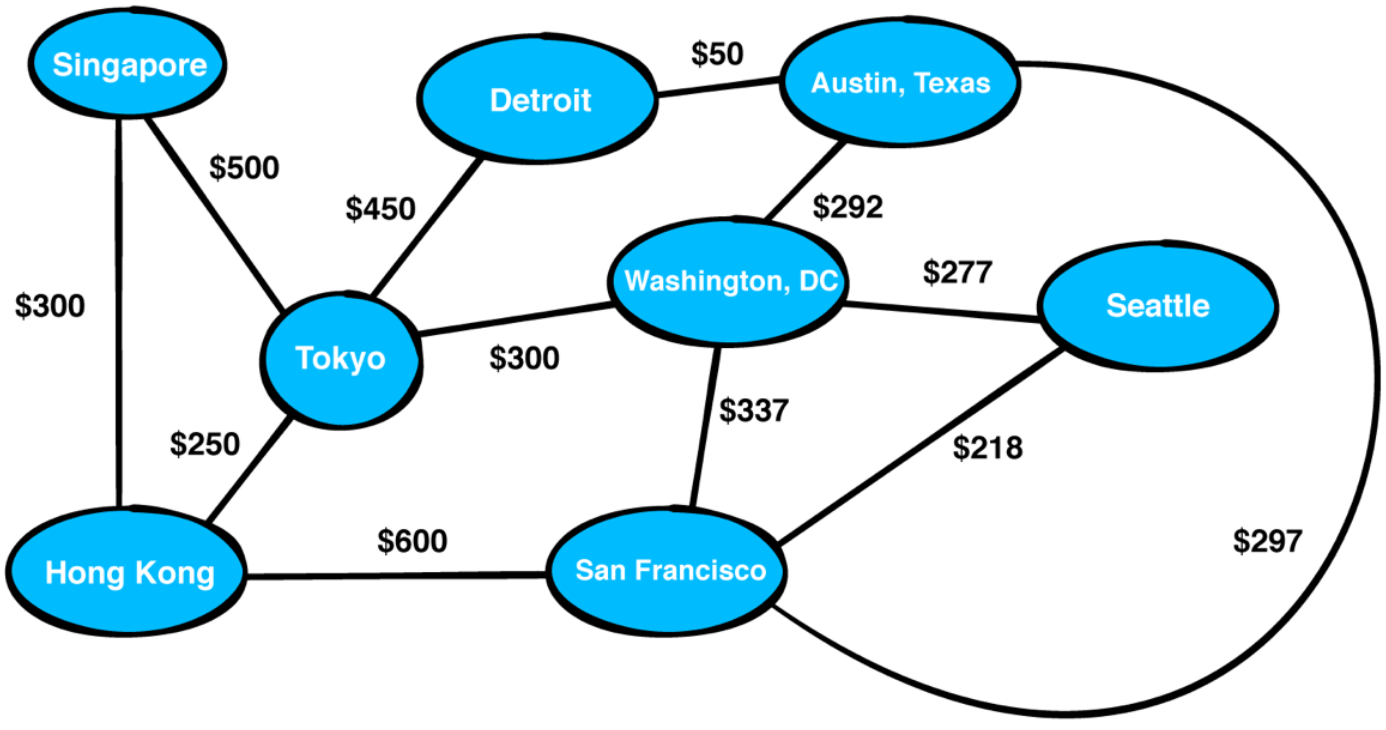
```
        }
      }
    }
  }
  val edgesDescription = grid.joinToString("\n")

  // 3
  return "$verticesDescription\n\n$edgesDescription"
}
```

Here are the steps:

1. You first create a list of the vertices.
2. Then, you build up a grid of weights, row by row.
3. Finally, you join both descriptions together and return them.

Building a network

You'll reuse the same example from `AdjacencyList`:



Go to `main()` and replace:

```
val graph = AdjacencyList<String>()
```

With:

```
val graph = AdjacencyMatrix<String>()
```

`AdjacencyMatrix` and `AdjacencyList` have the same interface, so the rest of the code stays the same.

You'll get the following output in your console:

```
0: Singapore
1: Tokyo
2: Hong Kong
3: Detroit
4: San Francisco
5: Washington DC
6: Austin Texas
7: Seattle

ø       500.0   300.0   ø       ø       ø       ø       ø
500.0   ø       250.0   450.0   ø       300.0   ø       ø
300.0   250.0   ø       ø       600.0   ø       ø       ø
ø       450.0   ø       ø       ø       ø       50.0    ø
ø       ø       600.0   ø       ø       337.0   297.0   218.0
ø       300.0   ø       ø       337.0   ø       292.0   277.0
ø       ø       ø       50.0    297.0   292.0   ø       ø
ø       ø       ø       ø       218.0   277.0   ø       ø

San Francisco Outgoing Flights:
-------------------------------
from: San Francisco to: Hong Kong
from: San Francisco to: Washington, DC
from: San Francisco to: Austin, Texas
from: San Francisco to: Seattle
```

In terms of visual beauty, an adjacency list is a lot easier to follow and trace than an adjacency matrix. Next, you'll analyze the common operations of these two approaches and see how they perform.

Graph analysis

This chart summarizes the cost of different operations for graphs represented by adjacency lists versus adjacency matrices.

Operations	Adjacency List	Adjacency Matrix
Storage Space	0(V + E)	0(V²)
Add Vertex	0(1)	0(V²)
Add Edge	0(1)	0(1)
Finding Edges and Weight	0(V)	0(1)

V represents vertices, and **E** represents edges.

An adjacency list takes less storage space than an adjacency matrix. An adjacency list simply stores the number of vertices and edges needed. As for an adjacency matrix, recall that the number of rows and columns is equal to the number of vertices. This explains the quadratic space complexity of $O(V^2)$.

Adding a vertex is efficient in an adjacency list: Simply create a vertex and set its key-value pair in the map. It's amortized as $O(1)$. When adding a vertex to an adjacency matrix, you're required to add a column to every row and create a new row for the new vertex. This is at least $O(V)$, and if you choose to represent your matrix with a contiguous block of memory, it can be $O(V^2)$.

Adding an edge is efficient in both data structures, as they are both constant time. The adjacency list appends to the array of outgoing edges. The adjacency matrix sets the value in the two-dimensional array.

Adjacency list loses out when trying to find a particular edge or weight. To

find an edge in an adjacency list, you must obtain the list of outgoing edges and loop through every edge to find a matching destination. This happens in *O*(V) time. With an adjacency matrix, finding an edge or weight is a constant time access to retrieve the value from the two-dimensional array.

Which data structure should you choose to construct your graph?

If there are few edges in your graph, it's considered a **sparse** graph, and an adjacency list would be a good fit. An adjacency matrix would be a bad choice for a sparse graph, because a lot of memory will be wasted since there aren't many edges.

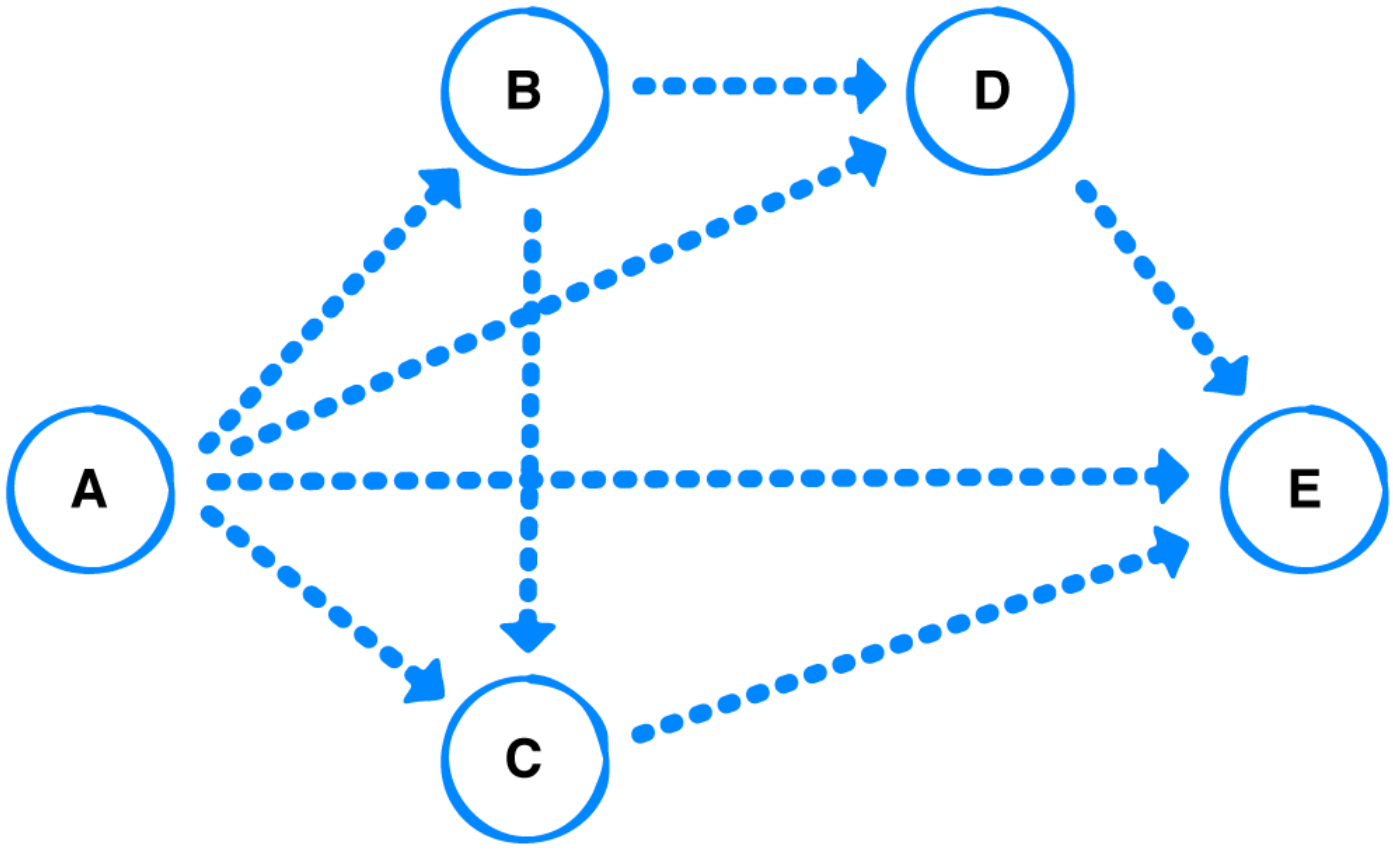
If your graph has lots of edges, it's considered a **dense graph**, and an adjacency matrix would be a better fit as you'd be able to access your weights and edges far more quickly.

- Adjacency matrix uses a square matrix to represent a graph.
- Adjacency list is generally good for **sparse graphs**, when your graph has the least amount of edges.
- Adjacency matrix is generally suitable for **dense graphs**, when your graph has lots of edges.

Challenges

Challenge 1: Find the distance between 2 vertices

Write a method to count the number of paths between two vertices in a directed graph. The example graph below has 5 paths from **A** to **E**:



Solution 1

The goal is to write a function that finds the number of paths between two vertices in a graph. One solution is to perform a depth-first traversal and keep track of the visited vertices.

```
fun numberOfPaths(
    source: Vertex<T>,
    destination: Vertex<T>
): Int {
  val numberOfPaths = Ref(0) // 1
  val visited: MutableSet<Vertex<Element>> = mutableSetOf() // 2
  paths(source, destination, visited, numberOfPaths) // 3
  return numberOfPaths.value
}
```

And to create a `Ref` class to pass the `Int` value by reference:

```
data class Ref<T: Any>(var value: T)
```

Here, you do the following:

1. `numberOfPaths` keeps track of the number of paths found between the `source` and `destination`.
2. `visited` is an `ArrayList` that keeps track of all the vertices visited.
3. `paths` is a recursive helper function that takes in four parameters. The first two parameters are the `source` and `destination` vertices. `visited` tracks the vertices visited, and `numberOfPaths` tracks the number of paths found. These last two parameters are modified within `paths`.

Add the following immediately after `numberOfPaths()`:

```
fun paths(
    source: Vertex<T>,
    destination: Vertex<T>,
    visited: MutableSet<Vertex<T>>,
    pathCount: Ref<Int>
) {
  visited.add(source) // 1
  if (source == destination) { // 2
    pathCount.value += 1
  } else {
    val neighbors = edges(source) // 3
    neighbors.forEach { edge ->
      // 4
      if (edge.destination !in visited) {
        paths(edge.destination, destination, visited, pathCount)
      }
    }
  }
  // 5
  visited.remove(source)
}
```

To get the paths from the `source` to `destination`:

1. Initiate the algorithm by marking the `source` vertex as visited.
2. Check to see if the `source` is the `destination`. If it is, you have found

a path, so increment the count by one.
3. If the destination has not be found, get all of the edges adjacent to the `source` vertex.
4. For every edge, if it has not been visited before, recursively traverse the neighboring vertices to find a path to the `destination` vertex.
5. Remove the `source` vertex from the visited list so that you can continue to find other paths to that node.

You're doing a depth-first graph traversal. You recursively dive down one path until you reach the destination, and back-track by popping off the stack. The time-complexity is *O*(V + E).

Challenge 2: The small world

Vincent has three friends: Chesley, Ruiz and Patrick. Ruiz has friends as well: Ray, Sun and a mutual friend of Vincent's. Patrick is friends with Cole and Kerry. Cole is friends with Ruiz and Vincent. Create an adjacency list that represents this friendship graph. Which mutual friend do Ruiz and Vincent share?

Solution 2

```
val graph = AdjacencyList<String>()

val vincent = graph.createVertex("vincent")
val chesley = graph.createVertex("chesley")
val ruiz = graph.createVertex("ruiz")
val patrick = graph.createVertex("patrick")
val ray = graph.createVertex("ray")
val sun = graph.createVertex("sun")
val cole = graph.createVertex("cole")
val kerry = graph.createVertex("kerry")

graph.add(EdgeType.UNDIRECTED, vincent, chesley, 0.0)
graph.add(EdgeType.UNDIRECTED, vincent, ruiz, 0.0)
graph.add(EdgeType.UNDIRECTED, vincent, patrick, 0.0)
graph.add(EdgeType.UNDIRECTED, ruiz, ray, 0.0)
```

```
graph.add(EdgeType.UNDIRECTED, ruiz, sun, 0.0)
graph.add(EdgeType.UNDIRECTED, patrick, cole, 0.0)
graph.add(EdgeType.UNDIRECTED, patrick, kerry, 0.0)
graph.add(EdgeType.UNDIRECTED, cole, ruiz, 0.0)
graph.add(EdgeType.UNDIRECTED, cole, vincent, 0.0)

println(graph)
println("Ruiz and Vincent both share a friend name Cole")
```

Key points

- You can represent real-world relationships through **vertices** and **edges**.
- Think of **vertices** as objects and **edges** as the relationship between the objects.
- Weighted graphs associate a weight with every edge.
- Directed graphs have edges that traverse in one direction.
- Undirected graphs have edges that point both ways.
- Adjacency list stores a list of outgoing edges for every vertex.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.