22 Dijkstra's Algorithm Written by Irina Galata

Have you ever used the Google or Apple Maps app to find the shortest distance or fastest time from one place to another? **Dijkstra's algorithm** is particularly useful in GPS networks to help find the shortest path between two places.

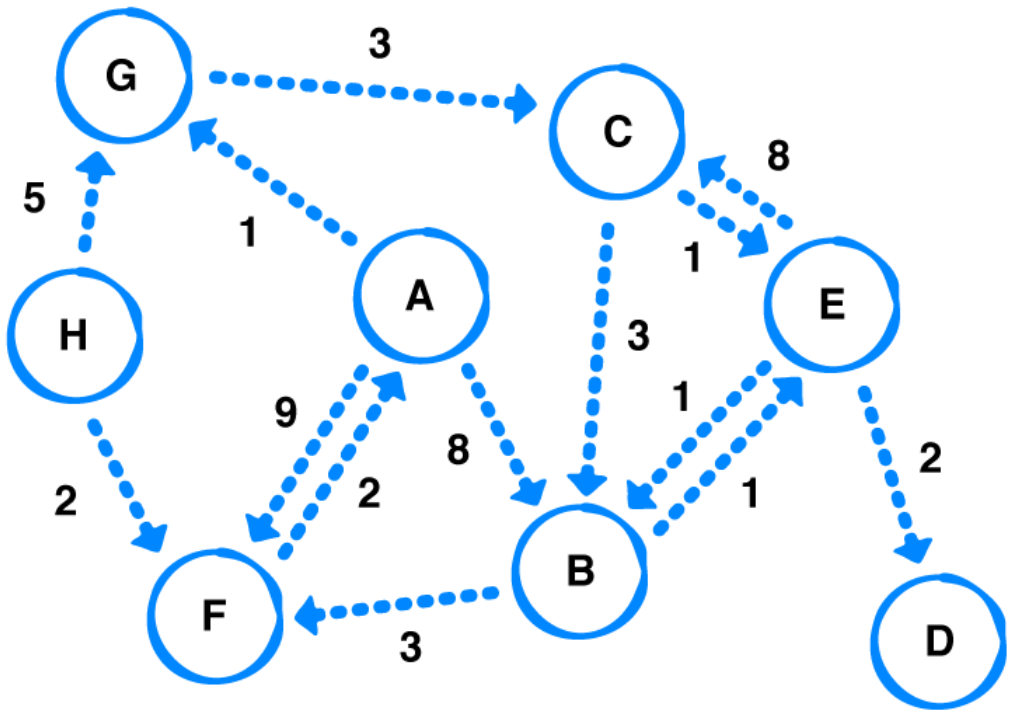
Dijkstra's algorithm is a greedy algorithm. A **greedy** algorithm constructs a solution step-by-step, and it picks the most optimal path at every step. In particular, Dijkstra's algorithm finds the shortest paths between vertices in either directed or undirected graphs. Given a vertex in a graph, the algorithm will find all shortest paths from the starting vertex.

Some other applications of Dijkstra's algorithm include:

1. Communicable disease transmission: Discover where biological diseases are spreading the fastest.
2. Telephone networks: Routing calls to highest-bandwidth paths available in the network.
3. Mapping: Finding the shortest and fastest paths for travelers.

Example

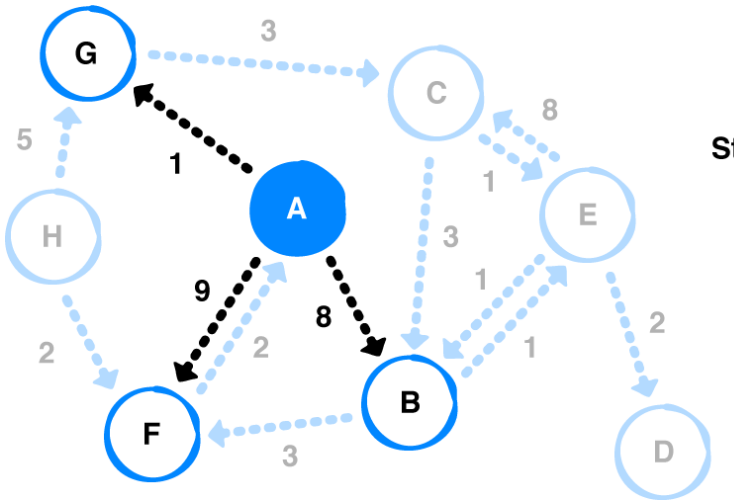
All the graphs you've looked at thus far have been undirected graphs. Let's change it up a little and work with a directed graph! Imagine the directed graph below represents a GPS network:



The vertices represent physical locations, and the edges between the vertices represent one way paths of a given cost between locations.

First pass

In Dijkstra's algorithm, you first choose a **starting vertex**, since the algorithm needs a starting point to find a path to the rest of the nodes in the graph. Assume the starting vertex you pick is **vertex A**.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null

From **vertex A**, look at all outgoing edges. In this case, you've three edges:

- **A** to **B**, has a cost of **8**.
- **A** to **F**, has a cost of **9**.
- **A** to **G**, has a cost of **1**.

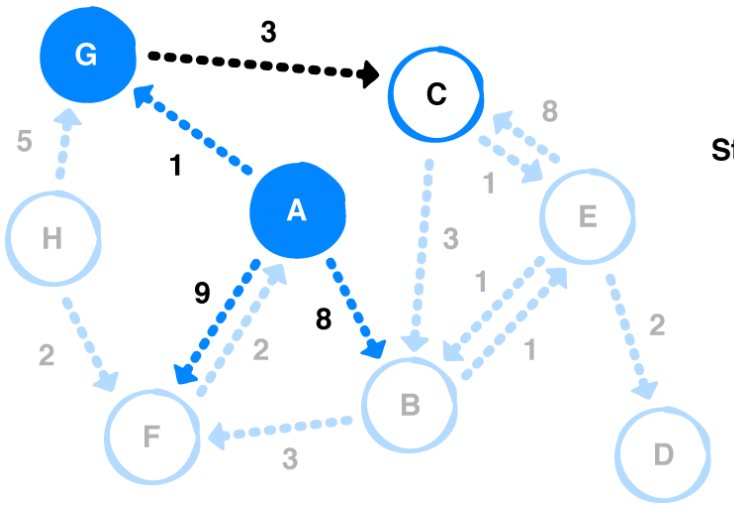
The remainder of the vertices will be marked as `null`, since there is no direct path to them from **A**.

As you work through this example, the table on the right of the graph will represent a history or record of Dijkstra's algorithm at each stage. Each pass of the algorithm will add a row to the table. The last row in the table will be the final output of the algorithm.

Second pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null

In the next cycle, Dijkstra's algorithm looks at the **lowest-cost** path you've thus far. **A** to **G** has the smallest cost of **1**, and is also the shortest path to get to **G**. This's marked with a dark fill in the output table.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null

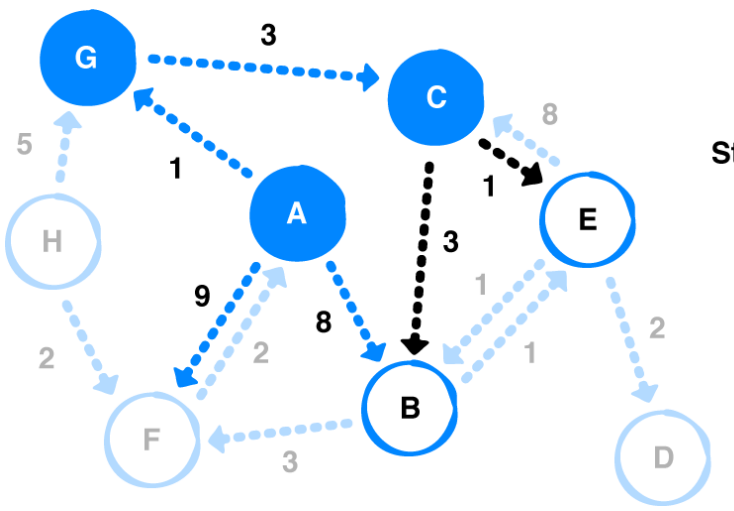
Now, from the lowest-cost path, **vertex G**, look at all the outgoing edges. There's only one edge from **G** to **C**, and its total cost is **4**. This is because the cost from **A** to **G** to **C** is **1 + 3 = 4**.

Every value in the output table has two parts: the total cost to reach that vertex, and the last neighbor on the path to that vertex. For example, the value **4 G** in the column for vertex **C** means that the cost to reach **C** is 4, and the path to **C** goes through **G**. A value of `null` indicates that no path has been discovered to that vertex.

Third pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null

In the next cycle, you look at the next-lowest cost. According to the table, the path to **C** has the smallest cost, so the search will continue from **C**. You fill column **C** because you've found the shortest path to get to **C**.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null

Look at all of **C**'s outgoing edges:

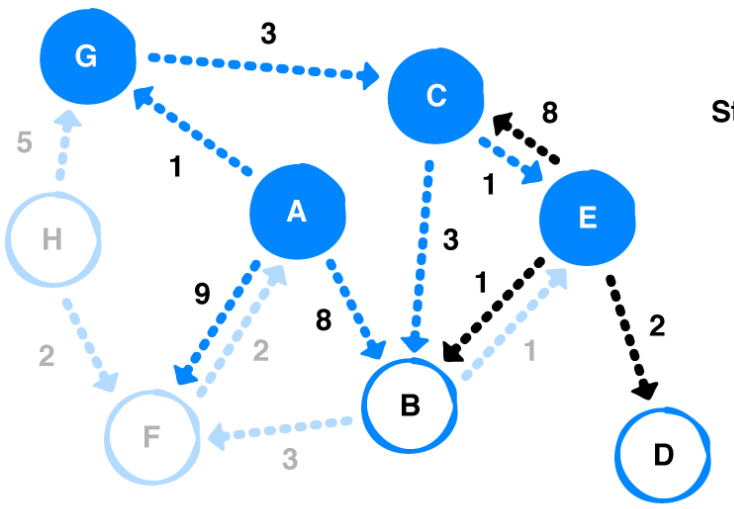
- **C** to **E** has a total cost of **4 + 1 = 5**.
- **C** to **B** has a total cost of **4 + 3 = 7**.

You've found a lower-cost path to **B**, so you replace the previous value for **B**.

Fourth pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null

Now, in the next cycle, ask yourself what is the next-lowest cost path? According to the table, **C** to **E** has the smallest total cost of **5**, so the search will continue from **E**.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null

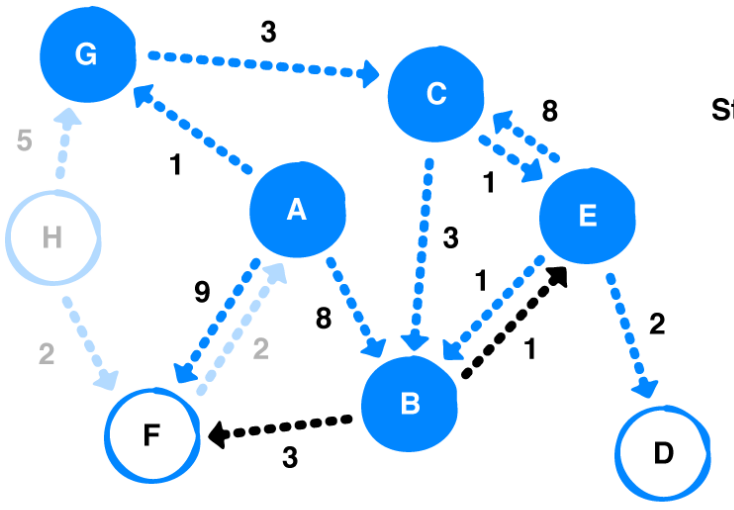
You fill column **E** because you've found the shortest path. Vertex **E** has the following outgoing edges:

- **E** to **C** has a total cost of **5 + 8 = 13**. Since you've found the shortest path to **C** already, disregard this path.
- **E** to **D** has a total cost of **5 + 2 = 7**.
- **E** to **B** has a total cost of **5 + 1 = 6**. According to the table, the current shortest path to **B** has a total cost of **7**. You update the shortest path from **E** to **B**, since it has a smaller cost of **6**.

Fifth pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null

Next, you continue the search from **B**.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null

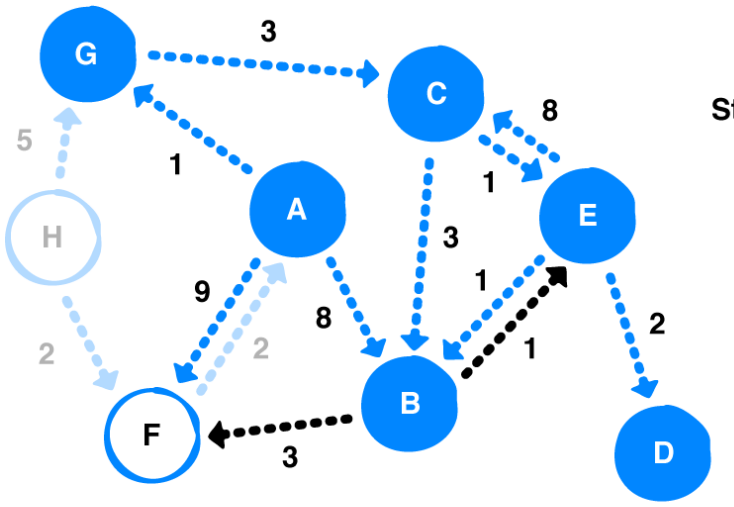
B has these outgoing edges:

- **B** to **E** has a total cost of **6 + 1 = 7**, but you've already found the shortest path to **E**, so disregard this path.
- **B** to **F** has a total cost of **6 + 3 = 9**. From the table, you can tell that the current path to **F** from **A** also has a cost of **9**. You can disregard this path since it isn't any shorter.

Sixth pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null

In the next cycle, you continue the search from **D**.



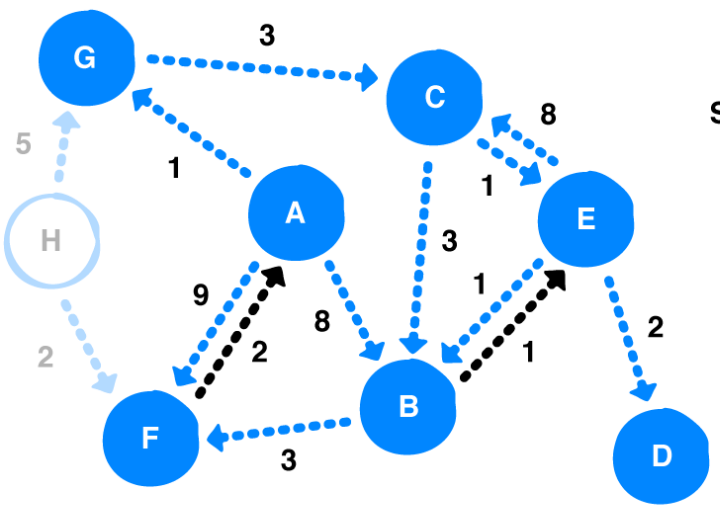
	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null
D	6 E	4 G	7 E	5 C	9 A	1 A	null

However **D** has no outgoing edges, so it's a dead end. You simply record that you've found the shortest path to **D** and move on.

Seventh pass

	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null
D	6 E	4 G	7 E	5 C	9 A	1 A	null

F is next up.

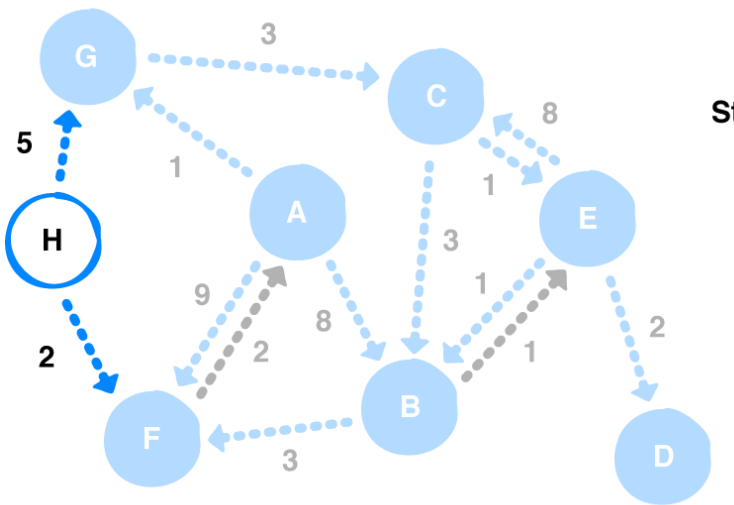


	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null
D	6 E	4 G	7 E	5 C	9 A	1 A	null
F	6 E	4 G	7 E	5 C	9 A	1 A	null

F has one outgoing edge to **A** with a total cost of **9 + 2 = 11**. You can disregard this edge since **A** is the starting vertex.

Eighth pass

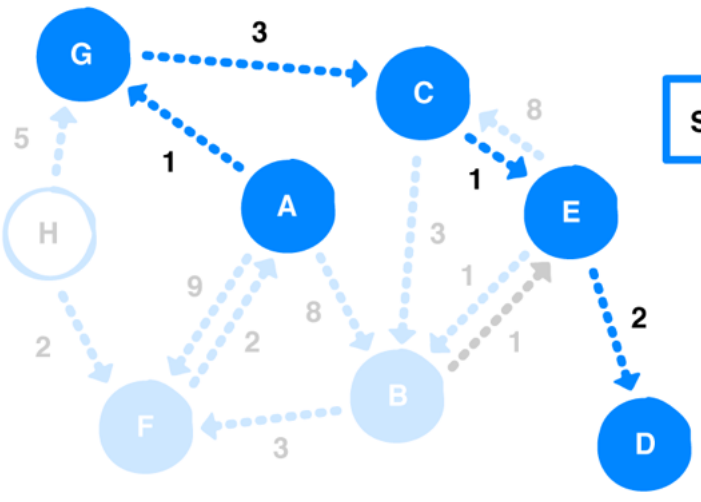
You've covered every vertex except for **H**. **H** has two outgoing edges to **G** and **F**. However, there's no path from **A** to **H**. This is why the whole column for **H** is `null`.



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	null
C	7 C	4 G	null	5 C	9 A	1 A	null
E	6 E	4 G	7 E	5 C	9 A	1 A	null
B	6 E	4 G	7 E	5 C	9 A	1 A	null
D	6 E	4 G	7 E	5 C	9 A	1 A	null
F	6 E	4 G	7 E	5 C	9 A	1 A	null

This completes Dijkstra's algorithm, since all the vertices have been visited!

Type something



	B	C	D	E	F	G	H
Start A	8 A	null	null	null	9 A	1 A	null
G	8 A	4 G	null	null	9 A	1 A	1 A
C	7 C	4 G	null	5 C	9 A	1 A	1 A
E	6 E	4 G	7 E	5 C	9 A	1 A	1 A
B	6 E	4 G	7 E	5 C	9 A	1 A	1 A
D	6 E	4 G	7 E	5 C	9 A	1 A	1 A
F	6 E	4 G	7 E	5 C	9 A	1 A	1 A

You can now check the final row for the shortest paths and their costs. For example, the output tells you the cost to get to **D** is **7**. To find the path, you simply backtrack. Each column records the previous vertex the current vertex is connected to. You should get from **D** to **E** to **C** to **G** and finally back to **A**. Let's look at how you can build this in code.

Implementation

Open up the starter project for this chapter. This project comes with an adjacency list graph and a priority queue, which you'll use to implement Dijkstra's algorithm.

The priority queue is used to store vertices that have not been visited. It's a min-priority queue so that, every time you dequeue a vertex, it gives you vertex with the current tentative shortest path.

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

```
class Dijkstra<T: Any>(private val graph: AdjacencyList<T>){
// to be continued ...
}
```

Next, add the following class bellow the `Dijkstra` class:

```
class Visit<T: Any>(val type: VisitType, val edge: Edge<T>? = null)

enum class VisitType {
  START, // 1
  EDGE // 2
}
```

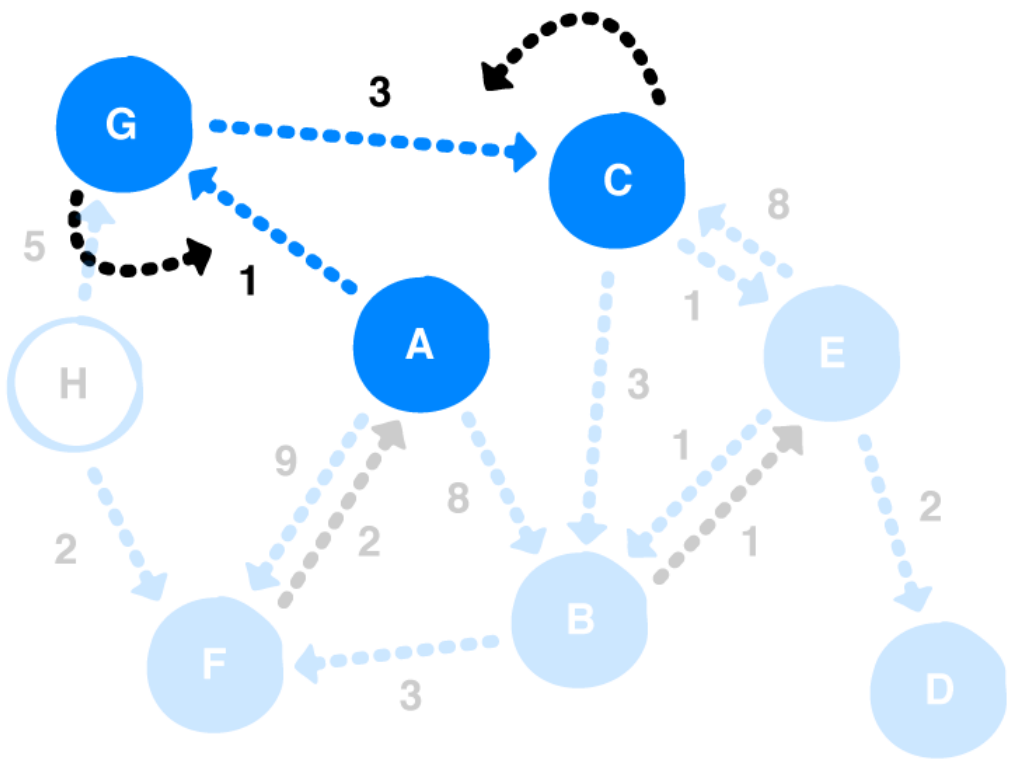
Here, you defined an enum named `Visit`. This keeps track of two states:

1. The vertex is the starting vertex.
2. The vertex has an associated `edge` that leads to a path back to the starting vertex.

Helper methods

Before building `Dijkstra`, let's create some helper methods that will help create the algorithm.

Tracing back to the start



C to G to A

You need a mechanism to keep track of the total weight from the current vertex back to the start vertex. To do this, you'll keep track of a map named `paths` that stores a `Visit` state for every vertex.

Add the following method to class `Dijkstra`:

```
private fun route(destination: Vertex<T>, paths: HashMap<Vertex<T>, Visit<T
  var vertex = destination // 1
  val path = arrayListOf<Edge<T>>() // 2

  loop@ while (true) {
    val visit = paths[vertex] ?: break

    when(visit.type) {
      VisitType.EDGE -> visit.edge?.let { // 3
```

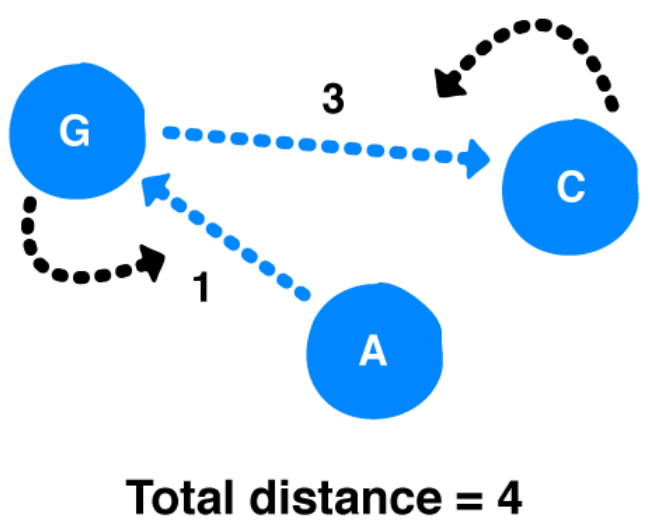
```
        path.add(it) // 4
        vertex = it.source // 5
      }
      VisitType.START -> break@loop // 6
    }
  }

  return path
}
```

This method takes in the `destination` vertex along with a dictionary of existing `paths`, and it constructs a path that leads to the `destination` vertex. Going over the code:

1. Start at the `destination` vertex.
2. Create a list of edges to store the path.
3. As long as you've not reached the `start` case, continue to extract the next `edge`.
4. Add this edge to the path.
5. Set the current vertex to the edge's `source` vertex. This moves you closer to the start vertex.
6. Once the `while` loop reaches the `start` case, you've completed the path and return it.

Calculating total distance



Total distance = 4

Once you've the ability to construct a path from the **destination** back to the **start** vertex, you need a way to calculate the total weight for that path.

Add the following method to class `Dijkstra`:

```
private fun distance(destination: Vertex<T>, paths: HashMap<Vertex<T>, Visi
  val path = route(destination, paths) // 1
  return path.sumByDouble { it.weight ?: 0.0 }
}
```

This method takes in the `destination` vertex and a dictionary of existing `paths`, and it returns the total weight. Going over the code:

1. Construct the path to the `destination` vertex.
2. `sumByDouble` sums the weights of all the edges.

Now that you've established your helper methods, let's implement Dijkstra's algorithm.

Generating the shortest paths

After the `distance` method, add the following:

```
fun shortestPath(start: Vertex<T>): HashMap<Vertex<T>, Visit<T>> {
  val paths: HashMap<Vertex<T>, Visit<T>> = HashMap()
  paths[start] = Visit(VisitType.START) // 1

  // 2
  val distanceComparator = Comparator<Vertex<T>>({ first, second ->
    (distance(second, paths) - distance(first, paths)).toInt()
  })
  // 3
  val priorityQueue = ComparatorPriorityQueueImpl(distanceComparator)
  // 4
  priorityQueue.enqueue(start)

  // to be continued ...
```

```
}
```

This method takes in a `start` vertex and returns a dictionary of all the paths. Within the method you:

1. Define `paths` and initialize it with the `start` vertex.
2. Create a `Comparator` which uses distances between vertices for sorting
3. Use the previous `Comparator` and create a min-priority queue to store the vertices that must be visited.
4. Enqueue the `start` vertex as the first vertex to visit.

Complete your implementation of `shortestPath` with:

```
while (true) {
  val vertex = priorityQueue.dequeue() ?: break // 1
  val edges = graph.edges(vertex) // 2

  edges.forEach {
    val weight = it.weight ?: return@forEach // 3

    if (paths[it.destination] == null
        || distance(vertex, paths) + weight < distance(it.destination, path
      paths[it.destination] = Visit(VisitType.EDGE, it)
      priorityQueue.enqueue(it.destination)
    }
  }
}

return paths
```

Going over the code:

1. You continue Dijkstra's algorithm to find the shortest paths until you've visited all the vertices have been visited. This happens once the priority queue is empty.

2. For the current `vertex`, you go through all its neighboring edges.
3. You make sure the edge has a weight. If not, you move on to the next edge.
4. If the `destination` vertex has not been visited before or you've found a cheaper path, you update the path and add the neighboring vertex to the priority queue.

Once all the vertices have been visited, and the priority queue is empty, you return the map of shortest paths back to the start vertex.

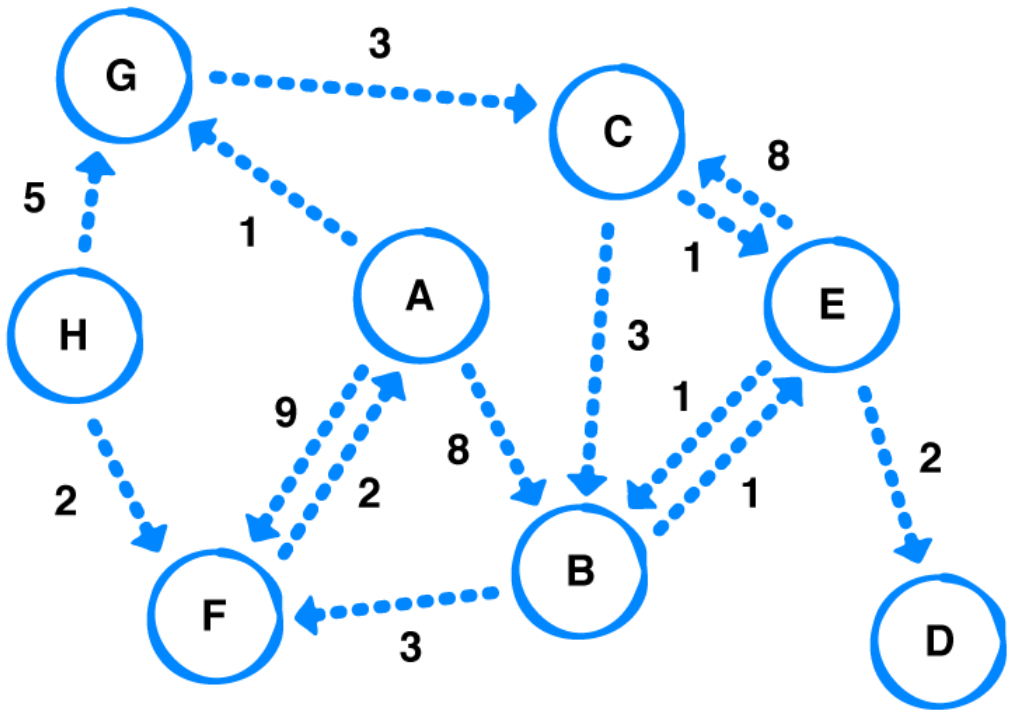
Finding a specific path

Add the following method to class `Dijkstra`:

```
fun shortestPath(destination: Vertex<T>, paths: HashMap<Vertex<T>,
  Visit<T>>): ArrayList<Edge<T>> {
    return route(destination, paths)
}
```

This simply takes the `destination` vertex and the map of shortest and returns the path to the `destination` vertex.

Trying out your code



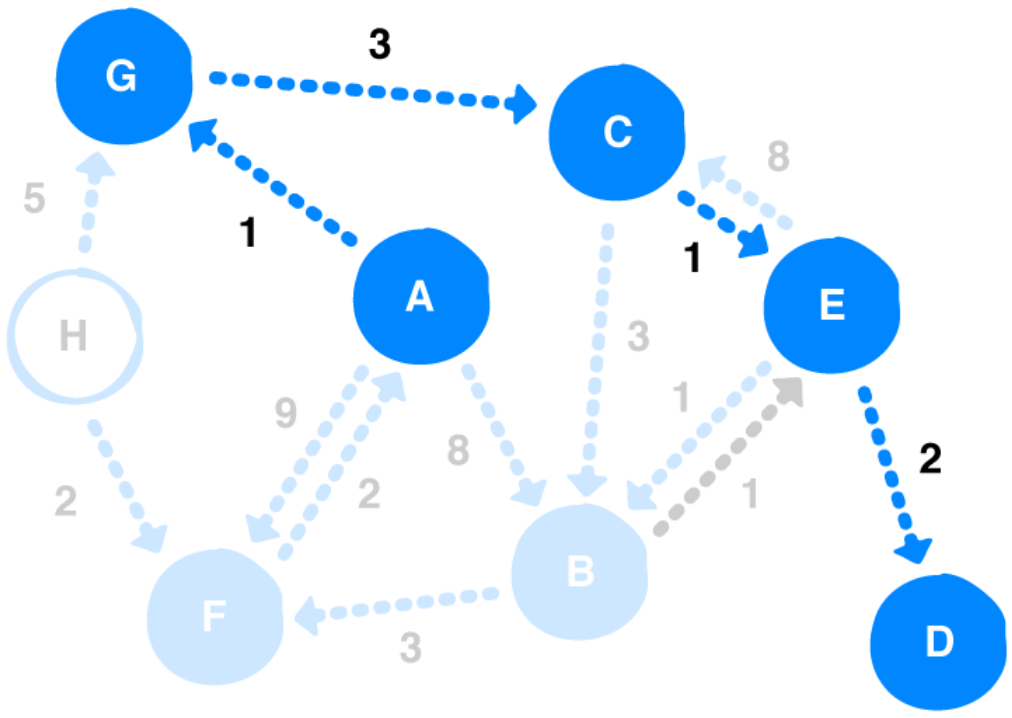
Navigate to the `main()` function, and you'll notice the graph above has been already constructed using an adjacency list. Time to see Dijkstra's algorithm in action.

Add the following code to the `main()` function:

```
val dijkstra = Dijkstra(graph)
val pathsFromA = dijkstra.shortestPath(a) // 1
val path = dijkstra.shortestPath(d, pathsFromA) // 2
path.forEach { // 3
 println("${it.source.data} --|${it.weight ?: 0.0}|--> + " +
            "${it.destination.data}")
}
```

Here, you simply create an instance of `Dijkstra` by passing in the graph network and do the following:

1. Calculate the shortest paths to all the vertices from the start vertex **A**.
2. Get the shortest path to **D**.
3. Print this path.



This outputs:

```
E --|2.0|--> D
C --|1.0|--> E
G --|3.0|--> C
A --|1.0|--> G
```

Performance

In Dijkstra's algorithm, you constructed your graph using an adjacency list. You used a min-priority queue to store vertices and extract the vertex with the minimum path. This has an overall performance of *O*(log V). This's because the heap operations of extracting the minimum element or inserting an element both take *O*(log V).

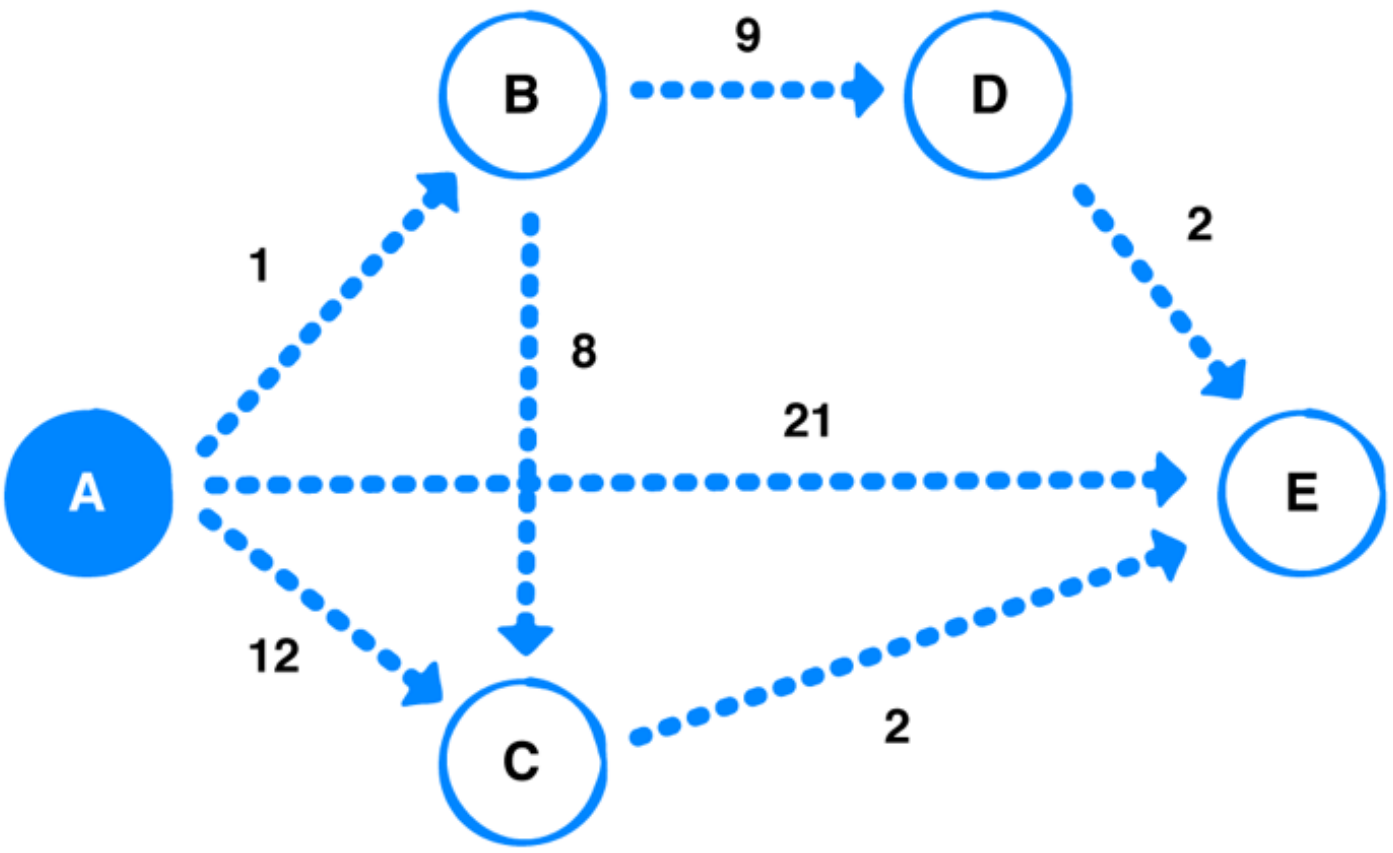
If you recall from the breadth-first search chapter, it takes *O*(V + E) to traverse all the vertices and edges. Dijkstra's algorithm is somewhat similar to breadth-first search, because you have to explore all neighboring edges. This time, instead of going down to the next level, you use a min-priority queue to select a single vertex with the shortest distance to traverse down. That means it is *O*(1 + E) or simply *O*(E). So, combining the traversal with operations on the min-priority queue, it takes

$O(E \log V)$ to perform Dijkstra's algorithm.

Challenges

Challenge 1: Running Dijkstra's

Given the following graph, step through Dijkstra's algorithm to produce the shortest path to every other vertex starting from **vertex A**. Provide the final table of the paths as shown in the previous chapter.



Solution 1

		B	C	D	E
Start	A	1 A	12 A	null	21 A
	B	1 A	9 B	10 B	21 A
	C	1 A	9 B	10 B	11 C
	D	1 A	9 B	10 B	11 C
	E	1 A	9 B	10 B	11 C

- Path to B: **A** - (1) - **B**
- Path to C: **A** - (1) - **B** - (8) - **C**
- Path to D: **A** - (1) - **B** - (9) - **D**
- Path to E: **A** - (1) - **B** - (8) - **C** - (2) - **E**

Challenge 2: Collect Dijkstra's data

Add a method to class `Dijkstra` that returns a dictionary of all the shortest paths to all vertices given a starting vertex. Here's the method signature to get you started:

```
fun getAllShortestPath(source: Vertex<T>): HashMap<Vertex<T>, ArrayList<Edg
  val paths = HashMap<Vertex<T>, Visit<T>>()

  // Implement solution here ...

  return paths
}
```

Solution 2

This function is part of **Dijkstra.kt**. To get the shortest paths from the `source` vertex to every other vertex in the graph, do the following:

```
fun getAllShortestPath(source: Vertex<T>): HashMap<Vertex<T>, ArrayList<Edg
  val paths = HashMap<Vertex<T>, ArrayList<Edge<T>>>() // 1
  val pathsFromSource = shortestPath(source) // 2

  graph.vertices.forEach { // 3
    val path = shortestPath(it, pathsFromSource)
    paths[it] = path
  }

  return paths // 4
}
```

1. The map stores the path to every vertex from the `source` vertex.
2. Perform Dijkstra's algorithm to find all the paths from the `source` vertex.
3. For every vertex in the graph, generate the list of edges between the `source` vertex to every vertex in the graph.
4. Return the map of paths.

Key points

- Dijkstra's algorithm finds a path to the rest of the nodes given a starting vertex.
- This algorithm is useful for finding the shortest paths between different endpoints.
- `Visit` state is used to track the edges back to the start vertex.
- The priority queue data structure helps to always return the vertex with the shortest path.
- Hence, it is a greedy algorithm!

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.