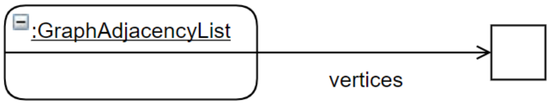
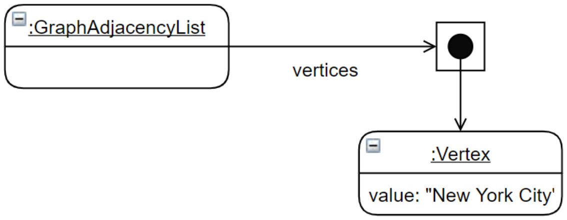
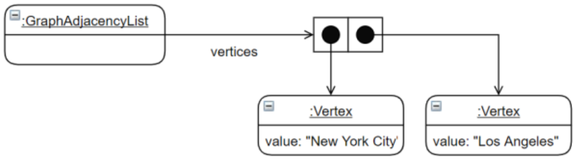
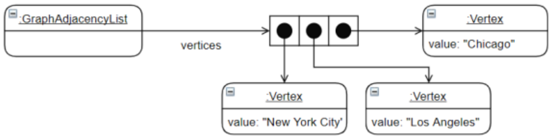
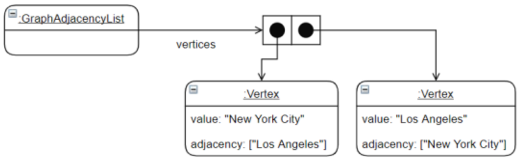
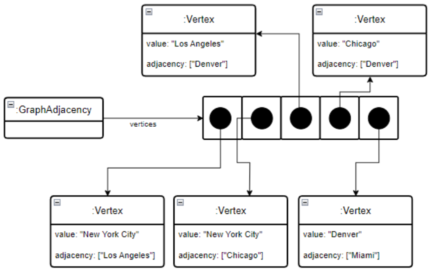
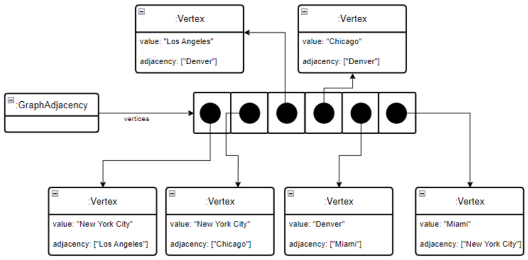
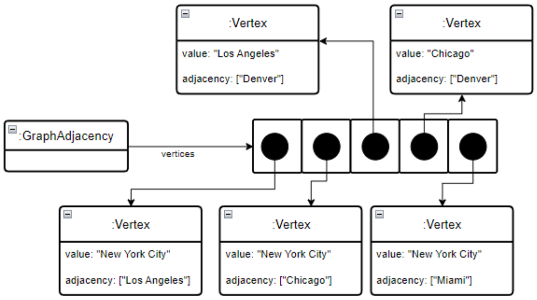
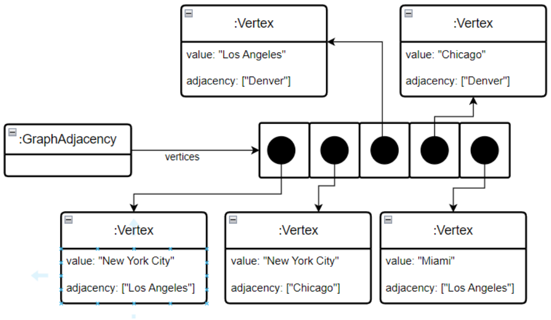
Configuration of GraphAdjacencyListTest scenarios

Name	Class	Scenery
setupScenary1	GraphAdjacencyListTest	
setupScenary2	GraphAdjacencyListTest	 :GraphAdjacencyList vertices :Vertex value: "New York City'
setupScenary3	GraphAdjacencyListTest	 :GraphAdjacencyList vertices :Vertex value: "New York City :Vertex value: "Los Angeles"
setupScenary4	GraphAdjacencyListTest	 :GraphAdjacencyList vertices :Vertex value: "Chicago" :Vertex value: "New York City' :Vertex value: "Los Angeles"
setupScenary5	GraphAdjacencyListTest	 :GraphAdjacencyList vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "Los Angeles" adjacency: ["New York City"]
setupScenary6	GraphAdjacencyListTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacency vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Denver" adjacency: ["Miami"]

setupScenary7	GraphAdjacencyListTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacency vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Denver" adjacency: ["Miami"] :Vertex value: "Miami" adjacency: ["New York City"]
setupScenary8	GraphAdjacencyListTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacency vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "New York City" adjacency: ["Miami"]
setupScenary9	GraphAdjacencyListTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacency vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Miami" adjacency: ["Los Angeles"]

Test Cases Design

Test objective: Test the correct operation of the GraphAdjacencyList class.

Class	Method	Scenery	Inputs	Result

GraphAdjacencyList	addVertex	setupScenary1	vertex = “New York City”	A new vertex with “New York City” as value is added to graph
GraphAdjacencyList	addVertex	setupScenary1	vertex1 = “New York City” vertex2 = “Los Angeles”	Two new vertices with “New York City” and “Los Angeles” as values are added to graph
GraphAdjacencyList	addVertex	setupScenary2	vertex = “New York City”	Vertex already exists exception is obtained

GraphAdjacencyList	addEdge	setupScenary3	source = “NewYork City”
			destination = “Los Angeles”
			weight = 1
				A new edge is added between the New York City and Los Angeles vertices.

GraphAdjacencyList	addEdge	setupScenary4	source = “NewYork City”
			destination = “Los Angeles”
				Two new edges are added, one between the New York City and Los Angeles vertices and

			weight = 3
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
				the other between the New York City and Chicago vertices.

GraphAdjacencyList	addEdge	setupScenary5	source = “NewYork City”
			destination = “Los Angeles”
			weight = 1
				Edge already exists exception is obtained

GraphAdjacencyList	removeVertex	setupScenary2	vertex = “New York City”	Remove a vertex from the graph
GraphAdjacencyList	removeVertex	setupScenary3	vertex1 = “New York City” vertex2 = “Los Angeles”	Remove two vertices from the graph

GraphAdjacencyList	removeVertex	setupScenary2	vertex = “New York City”	Exception for trying to remove a vertex that doesn't exist

GraphAdjacencyList	removeEdge	setupScenary5	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
				Remove a edge from the graph

GraphAdjacencyList	removeEdge	setupScenary4	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
				Remove two edges from the graph

GraphAdjacencyList	removeEdge	setupScenary2	vertex = “New York City”	Exception for trying to remove a vertex that doesn't exist

GraphAdjacencyList	BFS	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
				Verify that the implementation of BFS in the graph produces the expected distance from "New York City" in a graph

GraphAdjacencyList	BFS	setupScenary2	vertex = “New York City”	Verify that the BFS implementation properly handles the case of searching from a non-existent vertex and throws the appropriate exception in that scenario.

GraphAdjacencyList	BFS	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
				Verify that the implementation of BFS in the graph produces the expected distance from "New York City" in a cyclic graph.

			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3	

GraphAdjacencyList	DFS	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
				Verify that the implementation of DFS in the graph produces the expected distance from "New York City" in a graph

			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3	

GraphAdjacencyList	DFS	setupScenary2	vertex = “New York City”	Verify that the DFS implementation properly handles the case of searching from a non-existent vertex and throws the appropriate exception in that scenario.

GraphAdjacencyList	DFS	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
				Verify that the implementation of DFS in the graph produces the expected distance from

			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3
				"New York City" in a cyclic graph.

GraphAdjacencyList	dijkstra	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
				Verify that the implementation of Dijkstra in the graph produces the expected distance from "New York City" in a graph

GraphAdjacencyList	dijkstra	setupScenary2	vertex = “New York City”	Verify that the Dijkstra implementation properly handles the case of searching from a non-existent vertex and throws the appropriate exception in that scenario.

GraphAdjacencyList	dijkstra	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
				Verify that the implementation of DFS in the graph produces the expected distance from "New York City" in a cyclic graph.

			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3	

GraphAdjacencyList	floydWarshall	SetupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
				Verify that the implementation of the Floyd-Warshall algorithm in the graph produces the correct previous vertex in the shortest path from "New York City" to "Miami"

			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3	

GraphAdjacencyList	floydWarshall	SetupScenary8	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
				Verify that the implementation of the Floyd-Warshall algorithm in the graph with negative-weighted edges produces the correct previous vertex in the shortest path.

			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “New York City”
			destination = “Miami”
			weight = -10	

GraphAdjacencyList	floydWarshall	SetupScenary2	vertex = “New York City”	Verify that the implementation of the Floyd-Warshall algorithm properly handles the case of a graph with a single vertex and sets the value of the previous vertex as null

GraphAdjacencyList	prim	SetupScenary9	source = “NewYork City”
			destination = “Los Angeles”
			weight = 4
				The implementation of the Prim's algorithm in the graph finds a valid minimum

			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Mami”
			destination = “Los Angeles”
			weight = 1
				spanning tree, where the selected vertices have distances equal to 'weight'

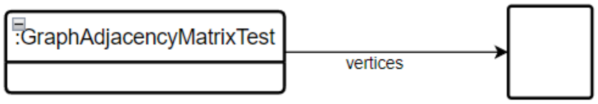
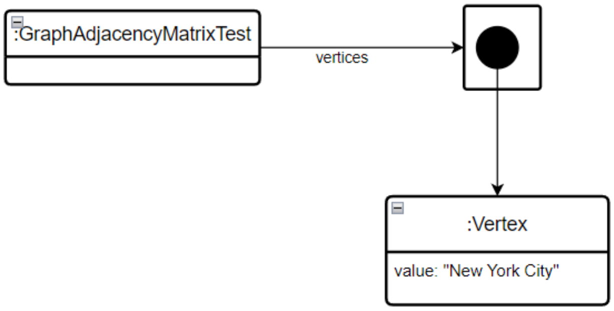
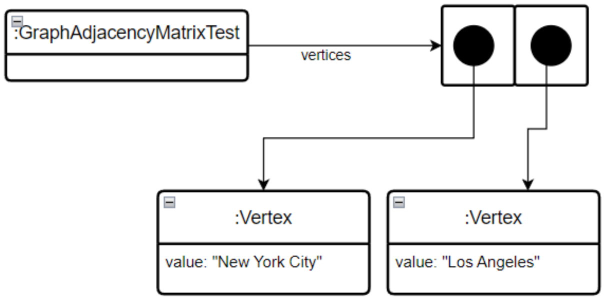
GraphAdjacencyList	prim	SetupScenary2	vertex = “New York City”	Verify that the Dijkstra implementation properly handles the case of searching from a non-existent vertex and throws the

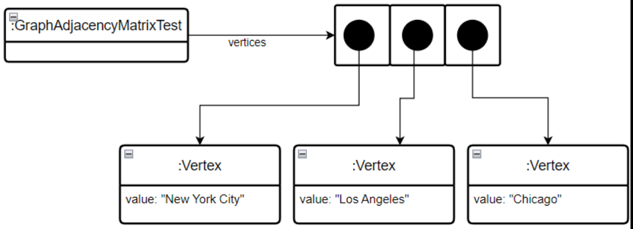
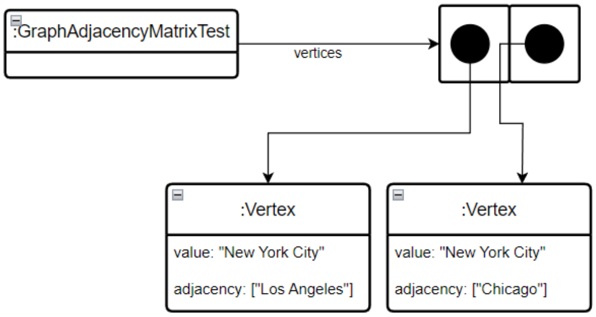
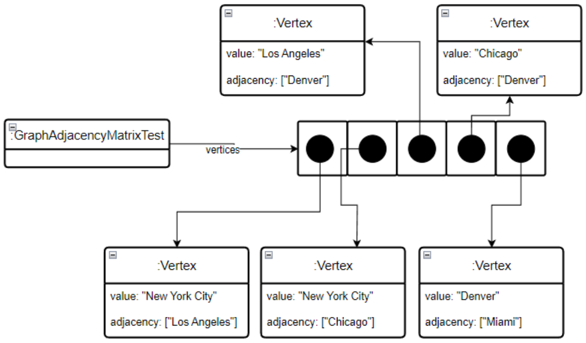
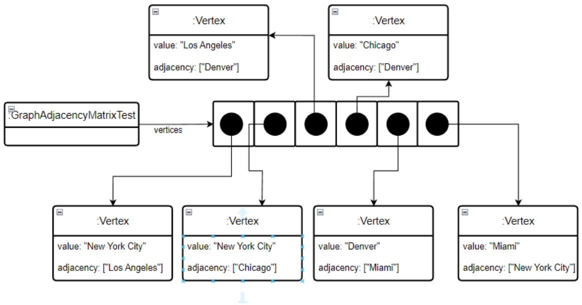
				appropriate exception in that scenario.

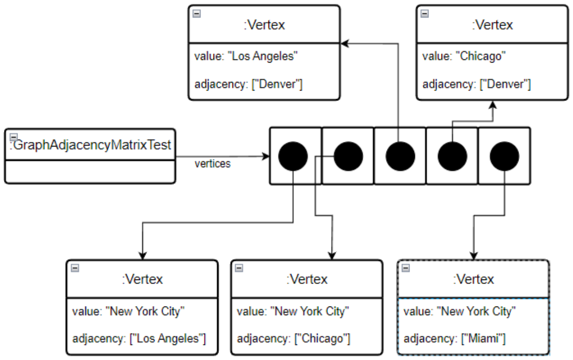
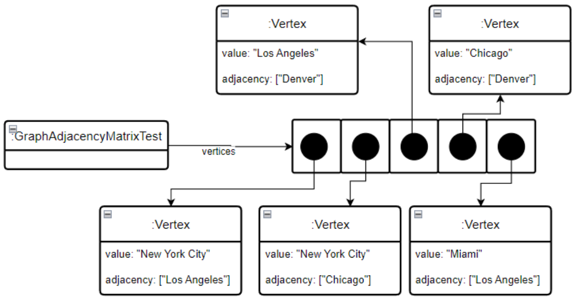
GraphAdjacencyList	prim	SetupScenary10	source = “NewYork City”
			destination = “Los Angeles”
			weight = -1
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Mami”
				Verify that the implementation of prim algorithm in the graph with negative-weighted edges produces the correct previous vertex in the shortest path

			destination = “Los Angeles”
			weight = -3	

Configuration of GraphAdjacencyMatrixTest scenarios

Name	Class	Scenery
setupScenary1	GraphAdjacencyMatrixTest	
setupScenary2	GraphAdjacencyMatrixTest	 :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City"
setupScenary3	GraphAdjacencyMatrixTest	 :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" :Vertex value: "Los Angeles"

setupScenary4	GraphAdjacencyMatrixTest	 :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" :Vertex value: "Los Angeles" :Vertex value: "Chicago"
setupScenary5	GraphAdjacencyMatrixTest	 :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"]
setupScenary6	GraphAdjacencyMatrixTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Denver" adjacency: ["Miami"]
setupScenary7	GraphAdjacencyMatrixTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Denver" adjacency: ["Miami"] :Vertex value: "Miami" adjacency: ["New York City"]

setupScenary8	GraphAdjacencyMatrixTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "New York City" adjacency: ["Miami"]
setupScenary9	GraphAdjacencyMatrixTest	 :Vertex value: "Los Angeles" adjacency: ["Denver"] :Vertex value: "Chicago" adjacency: ["Denver"] :GraphAdjacencyMatrixTest vertices :Vertex value: "New York City" adjacency: ["Los Angeles"] :Vertex value: "New York City" adjacency: ["Chicago"] :Vertex value: "Miami" adjacency: ["Los Angeles"]

Test Cases Design

Test objective: Test the correct operation of the GraphAdjacencyMatrixTest class.				
Class	**Method**	**Scenery**	**Inputs**	**Result**
GraphAdjacencyMatrixTest	addVertex	setupScenary1	vertex = “New York City”	A new vertex with “New York City” as value is added to graph

GraphAdjacencyMatrixTest	addVertex	setupScenary1	vertex1 = “New York City” vertex2 = “Los Angeles”	Two new vertices with “New York City” and “Los Angeles” as values are added to graph
GraphAdjacencyMatrixTest	addVertex	setupScenary2	vertex = “New York City”	Vertex already exists exception is obtained

GraphAdjacencyMatrixTest	addEdge	setupScenary3	source = “NewYork City”
			destination = “Los Angeles”
			weight = 1
				A new edge is added between the New York City and Los Angeles vertices.

GraphAdjacencyMatrixTest	addEdge	setupScenary4	source = “NewYork City”
			destination = “Los Angeles”
			weight = 1
			source = “NewYork City”
			destination = “Chicago”
				Two new edges are added, one between the New York City and Los Angeles vertices and the other between the New York City and Chicago vertices.

			weight = 2	

GraphAdjacencyMatrixTest	addEdge	setupScenary5	source = “NewYork City”
			destination = “Los Angeles”
			weight = 1
				Edge already exists exception is obtained

GraphAdjacencyMatrixTest	removeVertex	setupScenary2	vertex = “New York City”	Remove a vertex from the graph
GraphAdjacencyMatrixTest	removeVertex	setupScenary3	vertex1 = “New York City” vertex2 = “Los Angeles”	Remove two vertices from the graph
GraphAdjacencyMatrixTest	removeVertex	setupScenary2	vertex = “New York City”	Exception for trying to remove a vertex that doesn't exist

GraphAdjacencyMatrixTest	removeEdge	setupScenary5	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
				Remove an edge from the graph

GraphAdjacencyMatrixTest	removeEdge	setupScenary4	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
				Remove two edges from the graph

GraphAdjacencyMatrixTest	removeEdge	setupScenary2	vertex = “New York City”	Exception for trying to remove a vertex that doesn't exist

GraphAdjacencyMatrixTest	BFS	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
				Verify that the implementation of BFS in the graph produces the expected distance from

			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
				"New York City" in a graph

GraphAdjacencyMatrixTest	BFS	setupScenary2	vertex = “New York City”	Verify that the BFS implementation properly handles the case of searching from a non-existent vertex and

				throws the appropriate exception in that scenario.

GraphAdjacencyMatrixTest	BFS	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 4
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
				Verify that the implementation of BFS in the graph produces the expected distance from "New York City" in an acyclic graph.

			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3	

GraphAdjacencyMatrixTest	DFS	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
				Verify that the implementation of DFS in the graph produces the expected distance from "New York City" in a graph

			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3	

GraphAdjacencyMatrixTest	DFS	setupScenary2	vertex = “New York City”	Verify that the DFS implementation properly handles the case of searching from a non-existent vertex and throws the appropriate exception in that scenario.

GraphAdjacencyMatrixTest	DFS	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
				Verify that the implementation of DFS in the graph produces the expected distance from "New York City" in an acyclic graph.

			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3	

GraphAdjacencyMatrixTest	dijkstra	setupScenary6	source = “NewYork City”
			destination = “Los Angeles”
				Verify that the implementation of Dijkstra in the graph produces the expected

			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3
				distance from "New York City" in a graph

GraphAdjacencyMatrixTest	dijkstra	setupScenary2	vertex = “New York City”	Verify that the Dijkstra implementation properly handles the case of searching from a non-existent vertex and

				throws the appropriate exception in that scenario.

GraphAdjacencyMatrixTest	dijkstra	setupScenary7	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Denver”
				Verify that the implementation of DFS in the graph produces the expected distance from "New York City" in an acyclic graph.

			destination = “Miami”
			weight = 3
			source = “Miami”
			destination = “New York City”
			weight = 3	

GraphAdjacencyMatrixTest	floydWarshall	SetupScenary6	source = “NewYork City”
			destination = “Los Angeles”
			weight = 5
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
				Verify that the implementation of the Floyd-Warshall algorithm in the graph produces the correct previous vertex in the shortest path from "New York City" to "Miami"

			destination = “Denver”
			weight = 5
			source = “Denver”
			destination = “Miami”
			weight = 3	

GraphAdjacencyMatrixTest	floydWarshall	SetupScenary8	source = “NewYork City”
			destination = “Los Angeles”
			weight = 2
			source = “NewYork City”
			destination = “Chicago”
			weight = -4
			source = “Los Angeles”
			destination = “Denver”
			weight = 7
			source = “Chicago”
			destination = “Denver”
				Verify that the implementation of the Floyd-Warshall algorithm in the graph with negative-weighted edges produces the correct previous vertex in the shortest path.

			weight = 0
			source = “New York City”
			destination = “Miami”
			weight = -10	

GraphAdjacencyMatrixTest	floydWarshall	SetupScenary2	vertex = “New York City”	Verify that the implementation of the Floyd-Warshall algorithm properly handles the case of a graph with a single vertex and sets the value of the previous vertex as null

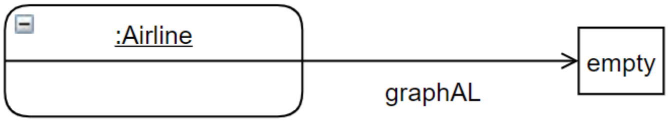
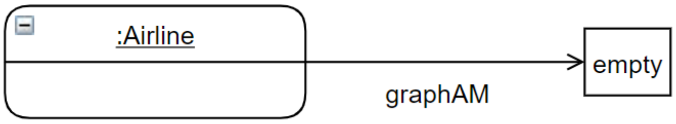
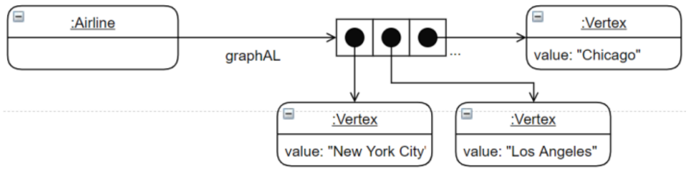
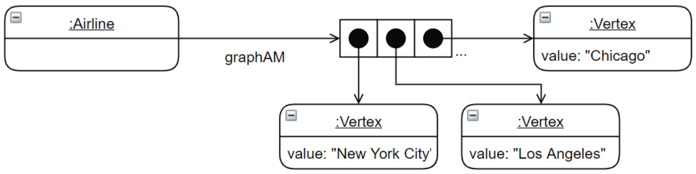
GraphAdjacencyMatrixTest	prim	SetupScenary9	source = “NewYork City”
			destination = “Los Angeles”
			weight = 4
			source = “NewYork City”
			destination = “Chicago”
				The implementation of the Prim's algorithm in the graph finds a valid minimum spanning tree, where the selected vertices have distances equal to 'weight'

			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Mami”
			destination = “Los Angeles”
			weight = 1	

GraphAdjacencyMatrixTest	prim	SetupScenary2	vertex = “New York City”	Verify that the Dijkstra implementation properly handles the case of searching from a non-existent vertex and throws the appropriate exception in that scenario.

GraphAdjacencyMatrixTest	prim	SetupScenary10	source = “NewYork City”
			destination = “Los Angeles”
			weight = -1
			source = “NewYork City”
			destination = “Chicago”
			weight = 2
			source = “Los Angeles”
			destination = “Denver”
			weight = 1
			source = “Chicago”
			destination = “Denver”
			weight = 5
			source = “Mami”
			destination = “Los Angeles”
			weight = -3
				Verify that the implementation of prim algorithm in the graph with negative-weighted edges produces the correct previous vertex in the shortest path

Configuration of AirlineTest scenarios

Name	Class	Scenery
setupScenary1	AirlineTest	
setupScenary2	AirlineTest	
setupScenary3	AirlineTest	 :Airline graphAL ... :Vertex value: "Chicago" :Vertex value: "New York City' :Vertex value: "Los Angeles"
setupScenary4	AirlineTest	 :Airline graphAM ... :Vertex value: "Chicago" :Vertex value: "New York City' :Vertex value: "Los Angeles"

Test Cases Design

Test objective: Test the correct operation of the Airline class.				
Class	**Method**	**Scenery**	**Inputs**	**Result**

Airline	loadCities	setupScenary1	graphOption = 1	Load vertices to the Graph with Adjacency List
Airline	loadCities	setupScenary2	graphOption = 2	Load vertices to the Graph with Adjacency Matrix

Airline	loadConnections	setupScenary3	weightOption = 0
			graphOption = 1
				Load connections to the graph with adjacency list, and time as weight

Airline	loadConnections	setupScenary4	weightOption = 0
			graphOption = 2
				Load connections to the graph with adjacency matrix, and time as weight

Airline	loadConnections	setupScenary3	weightOption = 1
			graphOption = 1
				Load connections to the graph with adjacency list, and cost as weight

Airline	loadConnections	setupScenary4	weightOption = 1
			graphOption = 2
				Load connections to the graph with adjacency matrix, and cost as weight

Airline	optimize	setupScenary3	weightOption = 0
			graphOption = 1
				Optimize connections depending on the time in the graph with adjacency list.

Airline	optimize	setupScenary3	weightOption = 1
			graphOption = 1
				Optimize connections depending on the cost in the graph with adjacency list.

Airline	optimize	setupScenary4	weightOption = 0
			graphOption = 2
				Optimize connections depending on the time in the graph with adjacency matrix.

Airline	optimize	setupScenary4	weightOption = 1
			graphOption = 2
				Optimize connections depending on the cost in the graph with

				adjacency matrix.