Test Design

PathFinder			
Method	Scenario	Input	Output
dijkstraPath	The graph contains three connected nodes, and the algorithm should select the shortest weighted path	Nodes: A, B, C Edges: A→B (1), B→C (1), A→C (10)	[A, B, C]
dijkstraPath	The destination node is not connected to the source node.	Nodes: A, B, C Edges: A→B (1)	An empty list ([])
dijkstraPath	There is a direct path from source to destination, but a cheaper path exists through an intermediate node	Nodes: A, B, C Edges: A→C (10), A→B (2), B→C (2)	[A, B, C]
dijkstraPathMatrix	the matrix contains three connected nodes. The algorithm should find the path with the smallest total weight from the start node to the destination	Nodes: A, B, C Edges: A→B (1), B→C (1), A→C (10)	[A, B, C]
dijkstraPathMatrix	The start and end node are the same, with no additional edges	Node: A	[A]
dijkstraPathMatrix	the algorithm should choose a longer but cheaper path over a direct but more expensive one	Nodes: A, B, C Edges: A→C (10), A→B (1), B→C (1)	[A, B, C]

Test Design

bfsPath	The graph contains two different paths of equal length. The algorithm should return a valid breadth-first path from source to target.	Nodes: A, B, C, D Edges: A→B→D and A→C→D	A path of 3 nodes starting with A and ending with D
bfsPath	The destination node is not connected to the source node	Nodes: A, B, C Edges: A→B	An empty list ([])
bfsPath	The graph provides a clear path of minimal node count between the start and end	Nodes: A, B, C, D Edges: A→B→D and A→C→D	A path of 3 nodes from A to D

Test Design

Node			
Method	**Scenario**	**Input**	**Output**
Node	Create a node with normal coordinates and valid ID	id = "A", x = 10, y = 20	id = "A", x = 10, y = 20, isWalkable = true, isHacked = false
Node	Create a node with negative coordinates	id = "Neg", x = -5, y = -10	id = "Neg", x = -5, y = -10, isWalkable = true, isHacked = false
Node	Create a node with origin coordinates (0,0)	id = "Zero", x = 0, y = 0	id = "Zero", x = 0, y = 0, isWalkable = true, isHacked = false
setHacked	Change hacked state from false to true	call setHacked(true) on a clean node	isHacked = true
setHacked	Change hacked state back from true to false	setHacked(true), then setHacked(false)	isHacked = false
setHacked	Apply setHacked(true) multiple times	call setHacked(true) twice	isHacked = true
equals	Compare two nodes with same ID but different positions	node1: id = "C", x = 1, y = 1 node2: id = "C", x = 5, y = 5	node1.equals(node2)=true, same hashCode
equals	Compare two nodes with different IDs	node1: id = "F" node2: id = "G"	node1.equals(node2) = false
equals	Compare node to null and a non-node object	node.equals(null), node.equals("A")	false

Test Design

Graph			
Method	Scenario	Input	Output
addNode	Add a single node to the graph	Node: A	Node A is added to the graph
addNode	Try adding the same node twice	Node: A added two times	Only one instance of Node A is in the graph
addNode	Add multiple different nodes	Nodes: A, B	Graph contains both A and B
addEdge	Add edge between two already added nodes	Nodes: A, B Edge A→B (weight 5)	Edge is added correctly
addEdge	Add edge between two nodes not previously added	Edge A→B (weight 2), without adding nodes manually	Nodes A and B are added and edge is created
addEdge	Add multiple edges from one node to different destinations	Edges: A→B (1), A→C (3)	Both edges are correctly added
getNeighbors	Retrieve neighbors of a node with one connected node	Edge: A→B	getNeighbors(A) → [B]
getNeighbors	Retrieve neighbors from a node with no edges	Node: A	getNeighbors(A) → empty list
getNeighbors	Retrieve neighbors from a node with multiple edges	Edges: A→B, A→C	getNeighbors(A) → [B, C]
getEdgeWeight	Get weight of an existing edge	Edge: A→B (weight 7)	getEdgeWeight(A, B) → 7
getEdgeWeight	Request weight from unconnected nodes	Nodes A, D (no edge)	getEdgeWeight(A, D) → Infinity
getEdgeWeight	Request weights of multiple edges from same source	Edges: A→B (1), A→C (2)	getEdgeWeight(A, B) → 1, getEdgeWeight(A, C) → 2

Test Design

Edge			
Method	Scenario	Input	Output
getSource	Standard: Source is a different node from destination	Edge from A to B	Returns Node A
getSource	Limit: Source and destination are the same node	Edge from A to A	Returns Node A
getSource	Interesting: Different node types as source	Edge from C to B	Returns Node C
getDestination	Standard: Destination is distinct	Edge from A to B	Returns Node B
getDestination	Limit: Destination equals source	Edge from C to C	Returns Node C
getDestination	Interesting: Destination is a different node	Edge from A to C	Returns Node C
getWeight	Standard: Positive weight	Weight = 10	Returns 10
getWeight	Limit: Zero weight	Weight = 0	Returns 0
getWeight	Interesting: Negative weight	Weight = -5	Returns -5
isDirected	Standard: Directed edge	Directed = true	Returns true
isDirected	Limit: Undirected edge	Directed = false	Returns false
isDirected	Interesting: Compare directed and undirected edges	One directed, one undirected	Returns true for one, false for the other

Test Design

AdjacencyMatrix			
Method	Scenario	Input	Output
addNode	Standard: Add unique node	Add node A, B, C	Matrix contains A, B, C
addNode	Limit: Duplicate node	Add node A again	Size remains 3
addNode	Interesting: Add up to 80 nodes	Add nodes N0 to N79	List size ≤ 80
addEdge	Standard: Directed edge added	Add edge A→B (5, directed)	Weight from A to B = 5
addEdge	Interesting: Undirected edge added	Add edge A↔B (7, undirected)	Weight A→B and B→A = 7
addEdge	Limit: Add edge to unregistered node	Edge A→D, D not added	Throws IllegalArgumentException
getEdgeWeight	Standard: Check edge weight	Edge A→C (4)	Returns 4
getEdgeWeight	Limit: No edge between nodes	Query A→B without edge	Returns Integer.MAX_VALUE
getEdgeWeight	Interesting: Self-loop	Query A→A	Returns 0
getNeighbors	Standard: Neighbor exists	Edge A→B added	Returns B in neighbors of A
getNeighbors	Limit: No neighbors	Query node D with no edges	Returns empty list
getNeighbors	Interesting: Undirected neighbor	Edge A↔B added	B in A neighbors, A in B neighbors
clear	Standard: Clear and check size	Add edges, call clear()	Nodes list empty, weights reset
clear	Limit: Call clear twice	Call clear() two times	No error, still empty
clear	Interesting: Reuse matrix after clear	Call clear(), add D	Nodes contains only D

Test Design

GeneratorGraphMap			
Method	Scenario	Input	Output
getNodes	Standard: Node list should not be null	Call getNodes()	Returns a non-null list
getNodes	Limit: List contains 80 nodes	Call getNodes()	List size is 80
getNodes	Interesting: First and last expected nodes exist	Call getNodes()	Contains 'Corner1' and 'Corner80'
generateGraph	Standard: Graph generation returns object	Call generateGraph()	Returns non-null Graph object
generateGraph	Limit: Graph must contain 80 nodes	Call generateGraph()	Graph contains 80 nodes
generateGraph	Interesting: Corner1 and Corner2 are connected	Call generateGraph(); retrieve Corner1 and Corner2	Corner1 has Corner2 as neighbor and vice versa

Test Design

Hacker			
Method	Scenario	Input	Output
getCurrentPosition / getVisitedNodes	Initial position is set correctly when Hacker is created	Start node	Position is 'Start'; visitedNodes contains only 'Start'
moveTo	Moves to a regular node (not a key)	Node with ID 'Node2'	Current position is 'Node2'; visitedNodes contains 'Node2'; no keys collected
moveTo	Moves to a key node	Node with ID containing 'Key'	Current position is key node; visitedNodes and collectedKeys contain it
moveTo	Tries to move to null node	null	Position unchanged; visitedNodes and collectedKeys unchanged
hasKey	Check true case	Node with ID 'KEYroom'	Returns true
hasKey	Check false case	Node with ID 'Lab1'	Returns false
reset	Reset hacker state to new node	New start node	Current position is new start; visited contains only it; keys cleared
setCurrentPosition / setCollectedKeys / setVisitedNodes	Updates internal lists and position	Custom node and lists	Values updated to given inputs