TAD Graph			
Graph = (V, A)			
∀ (u, v, w) ∈ A: (u ∈ V) ∧ (v ∈ V) ∧ (w > 0)			
Primitive Operations			
Operation	**Input**	**Input**	**Type**
addNode	Node	Void	Modifier
addEdge	Edge	void	Modifier
getNodes	—	List	Observer
getEdgesFrom	Node	List	Observer
getNeighbors	Node	List	Observer
getEdgeWeight	Node s, Node d	double	Observer

〈addNode(Graph, Node)〉
〈Adds a new node to the graph〉
Precondition: 〈node ≠ null ∧ node ∉ V〉
Postcondition: 〈node ∈ V ∧ adjacencyList[node] = []〉

〈addEdge(Graph, Edge)〉
〈Adds an edge between two nodes〉
Precondition: 〈edge ≠ null ∧ edge.source ∈ V ∧ edge.destination ∈ V ∧ edge.weight > 0〉
Postcondition: 〈edge ∈ A ∧ edge ∈ adjacencyList[edge.source]〉

〈addEdge(Grafo, T, T, int)〉
〈Adds a new edge with weight between two vertices〉
{pre: 〈u ∈ V ∧ v ∈ V ∧ w > 0〉 }
{post: 〈isDirected ⇒ (u, v, w) ∈ A ∨ ¬isDirected ⇒ (u, v, w) ∈ A ∧ (v, u, w) ∈ A〉 }

〈getNodes(Graph)〉
〈Returns all nodes in the graph〉
Precondition: —
Postcondition: 〈returns V〉

〈getEdgesFrom(Graph, Node)〉
〈Returns all edges starting from the given node〉
Precondition: 〈node ∈ V〉
Postcondition: 〈returns A' ⊆ A where ∀ e ∈ A': e.source = node〉

〈getNeighbors(Graph, Node)〉
〈Returns all neighbors (adjacent nodes) of a given node〉
Precondition: 〈node ∈ V〉
Postcondition: 〈returns list of nodes such that ∃ edge ∈ A: edge.source = node ∧ edge.destination ∈ list〉

〈getEdgeWeight(Graph, Node, Node)〉
〈Returns the weight of the edge from source to destination〉
Precondition: 〈 ∃ edge ∈ A: edge.source = source ∧ edge.destination = destination〉
Postcondition: 〈returns edge.weight if found, otherwise ∞〉

TAD AdjacencyMatrix
AdjacencyMatrix = (V, A, M)
$\forall$ i, j $\in$ [0, SIZE):(i = j $\Rightarrow$ M[i][j] = 0) $\wedge$ (i $\neq$ j $\Rightarrow$ ($\exists$ w $\in$ $\mathbb{N}^+$: edge(i, j, w) $\in$ A $\Rightarrow$ M[i][j] = w) $\vee$ ($\neg\exists$ edge(i, j, w) $\in$ A $\Rightarrow$ M[i][j] = ∞))
Primitive Operations

Operation	**Input**	**Input**	**Type**
addNode	Node	Void	Modifier
addEdge	Edge	void	Modifier
getNodes	—	List	Observer
clear	—	void	Modifier
getNeighbors	Node	List	Observer
getEdgeWeight	Node s, Node d	double	Observer
getNodeIndex	Node	int	Observer

〈addNode(AdjacencyMatrix, Node)〉
Adds a new node to the graph if it doesn't already exist and there is space
Precondition: node $\neq$ null $\wedge$ node $\notin$ V $\wedge$ |V| < SIZE
Postcondition: node $\in$ V

⟨addEdge(AdjacencyMatrix, Edge)⟩
Adds a new edge between two registered nodes, with direction or bidirection depending on the edge
Precondition: $edge \neq null \land edge.source \in V \land edge.destination \in V \land edge.weight > 0$
Postcondition: $M[i][j] \leftarrow w \land \neg edge.isDirected() \Rightarrow M[j][i] \leftarrow w$

⟨getNodes(AdjacencyMatrix)⟩
Returns the list of all registered nodes
Precondition: —
Postcondition: returns V

⟨getNeighbors(AdjacencyMatrix, Node)⟩
Returns the list of adjacent nodes of the given node
Precondition: $node \in V$
Postcondition: returns $L = \{ nodes[j] \in V \mid M[i][j] \neq \infty \land M[i][j] \neq 0 \}$

⟨getEdgeWeight(AdjacencyMatrix, Node u, Node v)⟩
Returns the weight of the edge between two nodes
Precondition: $u \in V \land v \in V$
Postcondition: $(M[i][j] \neq \infty \Rightarrow$ returns $M[i][j]) \land (M[i][j] = \infty \Rightarrow$ returns $\infty)$

〈getNodeIndex(AdjacencyMatrix, Node)〉
Returns the index of a node in the lis
Precondition: —
Postcondition: (node $\in$ V $\Rightarrow$ returns i $\in \mathbb{N}$ such that nodes[i] = node) $\wedge$ (node $\notin$ V $\Rightarrow$ returns -1)

〈clear(AdjacencyMatrix)〉
Clears all nodes and resets the matrix
Precondition: —
Postcondition: V = $\emptyset$ $\wedge$ $\forall$ i, j $\in$ [0, SIZE): (i = j $\Rightarrow$ M[i][j] = 0) $\wedge$ (i $\neq$ j $\Rightarrow$ M[i][j] = ∞)

TAD Edge			
Edge = (source, destination, weight, isDirected)			
source ≠ null ∧ destination ≠ null ∧ weight > 0			
Primitive Operations			
Operation	**Input**	**Input**	**Type**
getSource	—	Node	Observer
getDestination	—	Node	Observer
getWeight	—	int	Observer
isDirected	—	boolean	Observer

〈getSource(Edge)〉
Returns the source node of the edge
Precondition: —
Postcondition: returns source ∈ Node

⟨getDestination(Edge)⟩
Returns the destination node of the edge
Precondition: —
Postcondition: returns destination ∈ Node

⟨getWeight(Edge)⟩
Returns the weight of the edge
Precondition: —
Postcondition: returns weight ∈ $\mathbb{N}^+$

⟨isDirected(Edge)⟩
Returns the list of adjacent nodes of the given node
Precondition: node ∈ V
Postcondition: returns true ⇔ the edge is directed (source → destination) returns false ⇔ the edge is undirected (source ↔ destination)

TAD Node			
Node = (id, x, y, isWalkable, isHacked)			
id≠null∧x∈R∧y∈R∧isWalkable∈B∧isHacked∈B			
Opreaciones Primitivas:			
Operación	**Entrada**	**Salida**	**Tipo**
getId	—	String	Observer
getX	—	double	Observer
getY	—	double	Observer
isWalkable	—	boolean	Observer
isHacked	—	boolean	Observer
setHacked	boolean	void	Modifier
equals	Object	boolean	Observer
hashCode	—	int	Observer

⟨getId(Node)⟩
Returns the identifier of the node
Precondition: —
Postcondition: returns id ∈ String ∧ id ≠ null

⟨getX(Node)⟩
Returns the x-coordinate
Precondition: —
Postcondition: returns x ∈ ℝ

⟨getY(Node)⟩
Returns the y-coordinate
.Precondition: —
Postcondition: returns y ∈ ℝ

〈isWalkable(Node)〉
Returns whether the node is walkable
Precondition: —
Postcondition: returns isWalkable $\in \mathbb{B}$

〈isHacked(Node)〉
Returns whether the node has been hacked
.Precondition: —
Postcondition: returns isHacked $\in \mathbb{B}$

〈setHacked(Node, boolean)〉
Sets the hacked status of the node
Precondition: hacked $\in \mathbb{B}$
Postcondition: isHacked = hacked

〈equals(Node, Object)〉
Compares two nodes by id
.Precondition: o $\neq$ null
Postcondition: returns true $\Leftrightarrow$ this.id = o.id

〈hashCode(Node)〉
Returns a hash code based on id
Precondition: —
Postcondition: returns hash(id)

TAD PathFinder

PathFinder = (dijkstraPath, bfsPath, reconstructPath)

∀ method ∈ {dijkstraPath, bfsPath}: startNode ≠ null ∧ endNode ≠ null ∧ startNode ∈ V ∧ endNode ∈ V

Opreaciones Primitivas:

Operación	**Entrada**	**Salida**	**Tipo**
dijkstraPath	Graph, Node start, Node end	List	Analizadora
bfsPath	Graph, Node start, Node end	List	Analizadora
reconstructPath	Map<Node, Node>, Node end	List	Auxiliar

〈dijkstraPath(PathFinder, Graph, Node start, Node end)〉

Finds the shortest path from start to end using Dijkstra’s algorithm

Precondition: start ≠ null ∧ end ≠ null ∧ start ∈ V ∧ end ∈ V

Postcondition:(path ≠ ∅ ∧ path[0] = start ∧ path[|path|-1] = end ∧ ∀ alternativePath: validPath(start, end) ⇒ totalWeight(path) ≤totalWeight(alternativePath)) ∨ (path = ∅ ∧ no path exists from start to end) 〈bfsPath(PathFinder, Graph, Node start, Node end)〉

Finds a path from start to end using Breadth-First Search

Precondition: start ≠ null ∧ end ≠ null ∧ start ∈ V ∧ end ∈ V

Postcondition:(path ≠ ∅ ∧ path[0] = start ∧ path[|path|-1] = end ∧ path contains the minimal number of edges between start and end)∨ (path = ∅ ∧ no path exists from start to end)

〈reconstructPath(PathFinder, Map<Node, Node> cameFrom, Node end)〉

Reconstructs a path from the end node by following the cameFrom map

Precondition: end ∈ domain(cameFrom) ∨ end = null

Postcondition: ∀ i ∈ [1, |path|-1]: cameFrom(path[i]) = path[i-1] ∧ path[|path|-1] = end