TADs for Integrative Task 2

Carol Andrea Mosquera (A00403934)
Samuel Navia Quiceno (A00405006)
Simón García (A00371828)

TAD Graph
Graph = {(V, E) where V represents vertices $\{v_1, v_2, \ldots, v_n\}$ and E represents edges $\{e_1, e_2, \ldots, e_n\}$ which connect edges $\{v_x, v_y\}$}.
{inv: For V, $v_i \neq v_j$ if $i \neq j$} {inv: If the graph allows loops, e_x can be $\{v_x, v_x\}$} {inv: If the graph allows parallel edges, the subset of edges in E, $\{e_1, e_2, \ldots, e_n\}$ can represent the same connection $\{v_x, v_y\}$} {inv: For a directed graph, e_x within E has an order such that $\{v_x,v_y\} \neq \{v_y, v_x\}$} {inv: For an undirected graph, e_x within E doesn't have an order meaning that $\{v_x,v_y\} = \{v_y, v_x\}$, and both representations must exist since navegability is not specified.} {inv: For E, $\{e_1, e_2, \ldots, e_n\}$, each edge has an associated weight w}
Primitive Operations: ● createGraph: Boolean x Boolean x Boolean -> Graph ● addVertex: Value -> Graph ● addEdge: Value x Value -> Graph ● removeVertex: Value -> Graph ● removeEdge: Value x Value -> Graph ● searchVertex: Value -> Vertex ● bFS: Value -> Graph ● dijkstra: Value -> Graph

createGraph(b_1, b_2, b_3)
Creates an empty graph with the 3 specified conditions: b_1 allows parallel edges, b_2 allows loops, b_3 is directed or not.
{pre: b_1, b_2, b_3 must be booleans}
{post: Graph = {V, E} where V contains {null} and E contains {null}}

addVertex(v)
Adds a vertex to the graph with value v.
{pre: v must be the same type of value as the Graph.}
{post: Graph = {V, E} where V contains {v} and E contains {null}}

addEdge(v_1, v_2)
Adds the edge e that connects vertices v_1 and v_2 which hold the values of v_1 and v_2.
{pre: v_1 and v_2 must be of the same type of the value as the Graph.}
{post: Graph = {V, E} where V contains {v_1, v_2} and E contains {e} which represent {v_1, v_2}} {post: GraphException if either v_1 or v_2 don't exist}

removeVertex(v)
Removes the vertex V from the Graph.
{pre: v must be of the same type of value as the Graph}
{post: Graph = {V, E} where V represents {v_1, v_2, …, v_n} - v and E represents {e_1, e_2, …, e_n}} {post: GraphException if v doesn't exist}

removeEdge(v_1, v_2)
Removes all edges that connect vertices v_1 and v_2 which hold the value of v_1 and v_2.
{pre: v_1 and v_2 must be of same type of value as the Graph}
{post: Graph = {V, E} where V represents {v_1, v_2, …, v_n} and E represents {e_1, e_2, …, e_n} - subset {e_1, e_2, …, e_n} that represents {v_1, v_2}}

searchVertex(v)
Retrieves the vertex which holds the value of v.
{pre: v must be of the same type of value as the Graph}
{post: Vertex} {post: null, if the vertex doesn't exist}

bFS(v)
Completes a Breadth-First Search along all connected vertices in the Graph.
{pre: v must be of the same type of value as the Graph}

{post: Graph = {V, E} where V represents {v_1, v_2, …, v_n} which all which are connected have an assigned color, predecessor and distance (number of vertices) relative to the root v and E represents {e_1, e_2, …, e_n} {post: GraphException, if the vertex doesn't exist}

dijkstra(v)
Calculates the shortest path from vertex v to the rest in the graph taking into account the weight of every edge.
{pre: v must be of the type of value as the Graph}
{post: Graph = {V, E} where V represents {v_1, v_2, …, v_n} which are all connected and have an assigned distance which represents the shortest path between v and v_x and a predecessor associated to the vertex by which v_x was found through the minimum relative edge weight and E represents {e_1, e_2, …, e_n} where each e_x has an associated weight w_x} {post: GraphExeception, if the vertex doesn't exist}