**Integrative Task #2 Computation and Discrete Structures 1**

**By: Juan Pablo Parra, Stick Martinez and Daniel Gonzalez**

**TAD**

|  |
| --- |
| **TAD Graph** |
| TAD = {ArrayList, Matrix} |
| {inv : vertex != nill && vertex >1 && edge connect to vertex} |
| Primitive Operations  Create Graph -> Void [Constructor]  AddVertex(vert) -> boolean [Modifier]  AddEdge(fromVertice, toVertice, weight) -> boolean [Modifier]  RemoveVertex(vert) -> boolean [Modifier]  RemoveEdge(fromVertice, toVertice) -> boolean [Modifier]  FindVertex(vert) -> Vertex [Analyzer]  Dijkstra(fromVertice, toVertice, weight) -> ArrayList [Analyzer]  Floyd() ->double[][] [Analyzer]  PrimL() ->GraphAdjacencyList [Analyzer]  PrimM() ->GraphAdjacencyMatrix [Analyzer]  KruskalL() -> GraphAdjacencyList [Analyzer]  KruskalM() -> GraphAdjacencyMatrix [Analyzer]  BFS(vert) ->Void [Analyzer]  DFS(vert): ->Void [Analyzer] |

|  |
| --- |
| Create Graph -> Void [Constructor]  “Creates a new graph with no vertices and no edges”  {pre : TRUE}  {post: Graph = {nill, nill} } |

|  |
| --- |
| AddVertex(vert) -> boolean  “Adds a new node to the graph [Modifier]”  {pre : The node to be added must not exist in the graph (Adjacency list and adjacency matrix}  {post: The node is added to the graph} |

|  |
| --- |
| AddEdge(start, end, weight) -> boolean  “Adds a new Edge [Modifier]”  {pre: The nodes of the edge must exist in the graph}  {post: The edge is added to the graph with the specified weight (if weighted)} |

|  |
| --- |
| RemoveVertex(vert) -> boolean  “Deletes a Vertex [Modifier]”  {pre: The node to be deleted must exist in the graph}  {post: The node and all associated edges are removed from the graph} |

|  |
| --- |
| RemoveEdge(start, end) -> boolean  “Deletes an Edge [Modifier]”  {pre: The nodes of the edge and the edge itself must exist in the graph}  {post: The edge is removed from the graph} |

|  |
| --- |
| FindVertex(vert) -> Vertex  “Finds a Vertex in the Graph [Analyzer]”  {pre: The graph is either represented as an adjacency list or an adjacency matrix}  {post: If the vertex exists in the graph, the corresponding vertex is returned; otherwise, a null or default value is returned} |

|  |
| --- |
| Dijkstra(omVertice, toVertice, weight) -> ArrayList  “Dijkstra's Shortest Path Algorithm”  {pre: The graph must be weighted and connected}  {post: The shortest distances from the start node to all other nodes are returned} |

|  |
| --- |
| Floyd() -> double[][]  “Floyd's All Pairs Shortest Path Algorithm”  {pre: The graph must be weighted}  {post: The matrix of minimum distances between all pairs of nodes is returned} |

|  |
| --- |
| PrimL() -> GraphAdjacencyList  “Prim's Minimum Spanning Tree Algorithm [Modifier]”  {pre: The graph must be weighted and connected, represented as an adjacency list}  {post: The minimum spanning tree of the graph is returned as an adjacency list} |

|  |
| --- |
| PrimM() -> GraphAdjacencyMatrixh  “Prim's Minimum Spanning Tree Algorithm [Modifier]”  {pre: The graph must be weighted and connected, represented as an adjacency matrix}  {post: The minimum spanning tree of the graph is returned as an adjacency matrix} |

|  |
| --- |
| KruskalL() -> GraphAdjacencyList  “Kruskal's Minimum Spanning Tree Algorithm [Modifier]”  {pre: The graph must be weighted, represented as an adjacency list}  {post: The minimum spanning tree of the graph is returned as an adjacency list} |

|  |
| --- |
| KruskalM() -> GraphAdjacencyMatrix  “Kruskal's Minimum Spanning Tree Algorithm [Modifier]”  {pre: The graph must be weighted, represented as an adjacency matrix}  {post: The minimum spanning tree of the graph is returned as an adjacency matrix} |

|  |
| --- |
| BFS(start) -> Void  “Breadth-First Search”  {pre: The start node must exist in the graph}  {post: A list of nodes in BFS order is returned} |

|  |
| --- |
| DFS(start) -> Void  “Depth-First Search”  {pre: The start node must exist in the graph}  {post: A list of nodes in DFS order is returned |