|  |
| --- |
| **TAD Graph** |
| Graph = {Vertices = 〈〉, Edges = 〈 }  *∀ n ∈ N*  *∀ p ^ q ∈ N*  *∀ i ^ j ^ k ^ l ∈ n*  *≠* |
| {inv*:*  *if it is simple graph the edges are unordered so, is the same as ,*  *if it is directed the edges are ordered so, is the different from ,*  *}* |
| * CreateGraph: Graph * AddVertex: Graph X Vertex Graph * AddEdge: Graph X Key (origin) X Key (end) X double (weight) Graph * RemoveVertex: Graph X Key (Vertex) Graph X Vertex * RemoveEdge: Graph X Key (origin) X Key (end) Graph * BFS: Graph X Key (origin) Graph * DFS: Graph Graph * GraphSize: Graph int * Dijkstra: Graph Graph X List * Floyd Warshall: Graph Graph X double [][] * Prim: Graph X vertex Graph |
| **CreateGraph [***Graph***] – Constructor**  *“Creates a new and empty Graph”.*  *{Pre: None}.*  *{Post: Returns a Graph without elements.}*  **AddVertex [***addVertex***] – Modifier**  *“Given an key and a value it is checked if the key is already in the graph, if it is not it adds the value to the graph in the form of a vertex.*  *{Pre: There must be a Graph, and the key must be of a compatible type}.*  *{Post: Returns the updated Graph}.*  **AddEdge [***addEdge***] - Modifier**  *“Given an origin key, end key and a weight it is first checked if the two key correspond to a vertex on the graph if that is the case it relates the vertex or the vertexes (Depending on the type of graph) with the value of the weight”.*  *{Pre: There must be a Graph, keys must be a compatible type, the weight must be congruent}.*  *{Post: Returns the updated Graph and the vertex with the new edge.}*  **RemoveVertex [***removeVertex***] - Modifier**  *“Removes and return the vertex with the key given, else it returns null, also it deletes the vertexs from the list of adjacencies in each vertex if the vertex removed is adjacent”.*  *{Pre: There must be a Graph}.*  *{Post: Returns the updated Graph and the vertex removed.}*  **RemoveEdge [***removeEdge***] - Modifier**  *“Removes the relation of the vertex or vertexes if they were adjacent proven by the given origin and end key, depending on the type of graph it may remove from the end vertex also”.*  *{Pre: There must be a Graph}.*  *{Post: Returns the updated Graph and vertexes.}*  **BFS [***bfs***] – Modifier**  *“It is a travel based on amplitude, it starts from a given vertex and discover each vertex adjacent to that one and all the adjacent to those, till all the vertices are discovered and linked by a parent in a type of three, which represents all the vertices reachable from the given vertex and it represents the shortest path from the origin”.*  *{Pre: There must be a Graph, the origin or given vertex must be on a highly connected graph, else it will be discovered only one part of the graph}.*  *{Post: Returns the updated Graph, with the parent link}.*  **DFS [***dfs***] – Modifier**  *“It is a depth search that travel ordered but not uniformly, that starts at the root vertex (selecting some arbitrary vertex as the root vertex) and explores as far as possible along each path before backtracking and repeating”.*  *{Pre: There must be a non-empty Graph (As it goes from all the vertices not visited it could be not highly connected)}.*  *{Post: Returns the updated Graph, with the parent link}.*  **Dijsktra [***dijsktra***] – Modifier & Analyzer.**  *“Dijkstra's algorithm starts from a source vertex and progressively explores the neighboring vertices, updating the shortest path distances from the source to each vertex. It maintains a priority queue to select the next vertex to explore based on the current known distances, it continues until all vertices are visited, making sure it the shortest paths from the source to all other vertices.”*  *{Pre: The graph must be a weighted graph and the source vertex or key must be compatible with the ones in the graph}*  *{Post: Returns the shortest path distances from the source vertex to all other vertices and the updated graph}*  **Floyd Warshall [***floydWarshall***] – Modifier & Analyzer.**  *“The Floyd-Warshall algorithm estimates the shortest paths between all pairs of vertices in a graph. It maintains a matrix of shortest path distances. This process is repeated until all pairs of vertices have been examined, resulting in a matrix containing the shortest path distances between every pair of vertices.”*  *{Pre: The graph must be a weighted graph}*  *{Post: Returns a matrix of shortest path distances between all pairs of vertices and the updated graph}*    **Prim [***prim***] – Modifier.**  *“Prim's algorithm grows a minimum spanning tree by selecting the edge with the smallest weight that connects a vertex in the current tree to a vertex outside the tree. It starts with an arbitrary vertex and incrementally adds vertices to the growing tree until all vertices are included. It never creates cycles, guaranteeing that the final result is a minimum spanning tree for the given graph.”*  *{Pre: The graph must be a connected, undirected graph and the vertex or its key must be of a compatible type}*  *{Post: Returns a minimum spanning tree for the given graph, the tree includes all vertices of the original graph and a subset of the edges and the updated graph}*  **Kruskal [***Kruskal***] – Modifier.**  *“Kruskal's algorithm finds a minimum spanning tree in a connected, undirected graph. The algorithm works by considering edges in ascending order of their weights and adding them to the minimum spanning tree if adding the edge doesn't create a cycle. It employs a disjoint-set data structure to track the connected components in the graph. Kruskal's algorithm continues this process until all vertices are included in the minimum spanning tree.”*  *{Pre: The graph must be a connected, undirected graph}*  *{Post: Returns a minimum spanning tree for the given graph and the tree includes all vertices of the original graph and a subset of the edges}*  **GraphSize [***size***] – Analyzer.**  *“It returns the number of vertexs with value on the graph.”.*  *{Pre: There must be a Graph}.*  *{Post: Returns a number corresponding to the operation}.* |