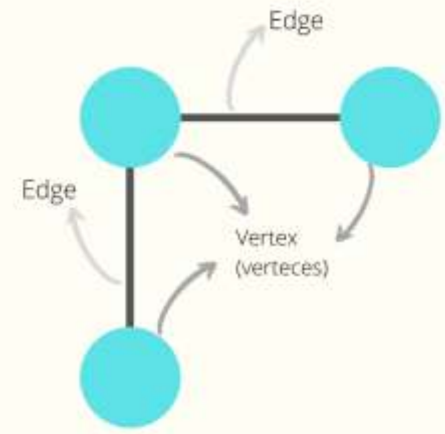
ADT Graph

ADT Graph <T>



Graph = { Directed = <directed>, Weighted = <weighted>, Vertices = <vertices>, Edges = <edges> }

{ inv: }

Primitive Operations:

• Graph:	directed x weighted	→ Graph<T>
• getVertices:	Graph<T>	→ List of vertices
• isDirected:	Graph<T>	→ Boolean
• isWeighted:	Graph<T>	→ Boolean
• addVertex:	Graph<T> x T	→ Graph<T>
• addEdge:	Graph<T> x Vertex<T> x Vertex<T>	→ Graph<T>
• removeVertex:	Graph<T> x Vertex<T>	→ Graph<T>
• removeEdge:	Graph<T> x Vertex<T> x Vertex<T>	→ Graph<T>
• getNeighborts:	Graph<T> x Vertex<T>	→ List of vertices
• getNumberOfVertices:	Graph<T>	→ Number
• getNumberOfEdge:	Graph<T>	→ Number
• areAdjacent:	Graph<T> x Vertex<T> x Vertex<T>	→ Boolean
• isInGraph:	Graph<T> x T	→ Boolean
• getEdgeWeight:	Graph<T> x Vertex<T> x Vertex<T>	→ double
• setEdgeWeight:	Graph<T> x Vertex<T> x Vertex<T> x double	→ Graph
• bfs:	Graph<T> x Vertex<T>	→ List of vertices
• dfs:	Graph<T> x Vertex<T>	→ List of vertices
• dijkstra:	Graph<T> x Vertex<T>	→ List of vertices
• floydWarshall:	Graph<T>	→ Matrix of double
• prim:	Graph<T> x Vertex<T>	→ Graph<T>
• kruskal:	Graph<T>	→ List of edges
• searchVertex:	Graph<T> x T	→ Vertex<T>
• getEdges:	Graph<T>	→ List of edges
• getContests:	Graph<T>	→ List of T

Graph(directed, weighted)

“Create a new Graph without edges”

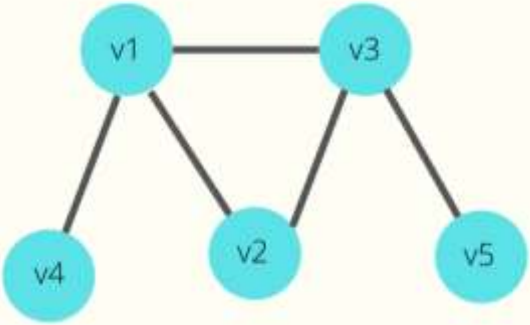
{ pre: TRUE ⋀ directed ∈ Boolean ⋀ weighted ∈ Boolean}

{ post: graph = { Directed = directed, Weighted = weighted, Vertices = 0, Edge = 0} }

getVertices(graph)

“Returns a collections of vertices”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }



{post: = {v1, v2, .. vn } n = Vertices}

v1
v2
v3
v4
v5

isDirected(graph)

“Returns the directed value”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }

{post: TRUE if graph is a directed graph
FALSE if graph is a undirected graph }

isWeighted(graph)

“Returns the weighted value”

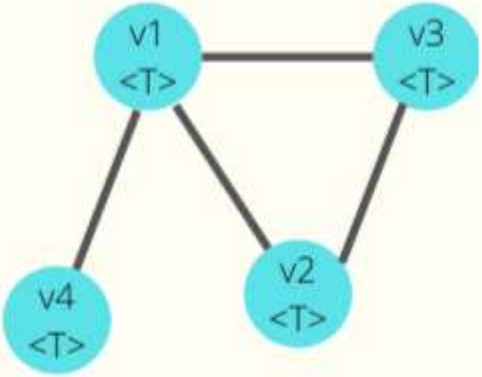
{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }

{post: TRUE if graph is a directed graph
FALSE if graph is a undirected graph }

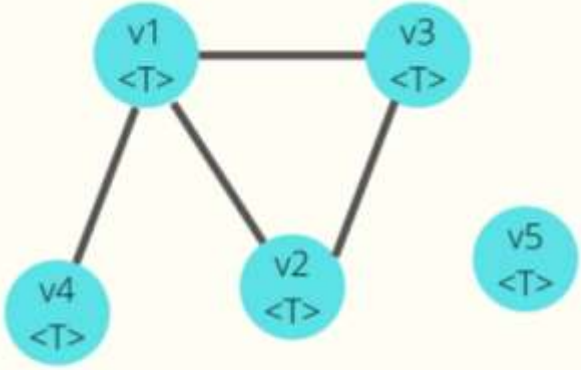
addVertex(graph, object)

“Add a new vertex to graph”

{pre: graph ={ …, Vertices = vertices, …. } ⋀ object ∈ T }



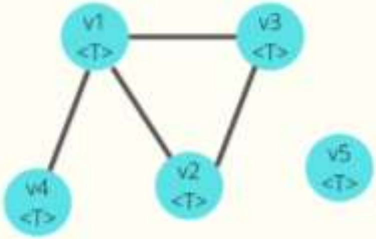
{post: graph ={ Directed = directed, Weighted = weighted, Vertices = vertices+1, Edge = edges } }



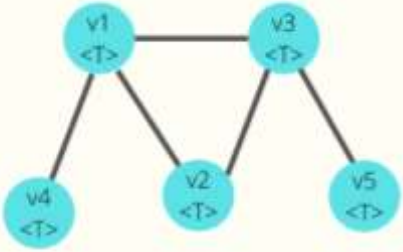
addEdge(graph, vertex1, vertex2)

"Add a new edge between two vertex of graph"

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph }



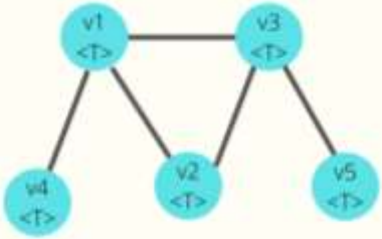
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges +1}}



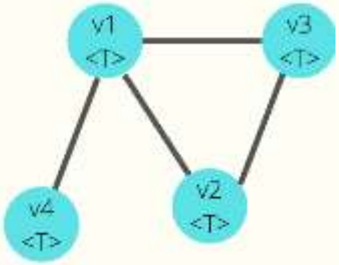
removeVertex(graph, vertex1)

"Remove a vertex of the graph"

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph }



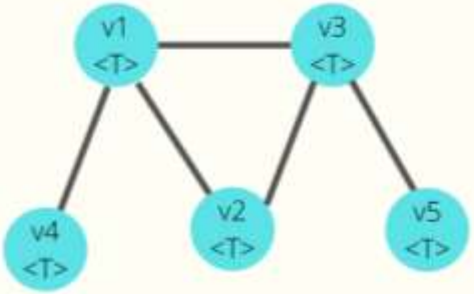
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices-1, Edge <=edges}}



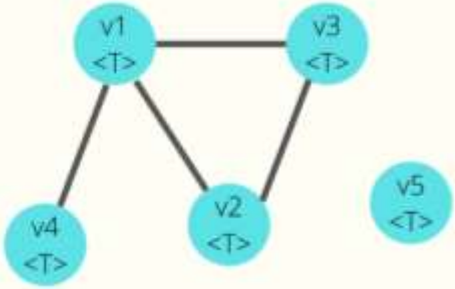
removeEdge(graph, vertex1, vertex2)

“Remove a connection between two vertices of graph”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
∧ vertex1 ∈ graph ∧ vertex2 ∈ graph }



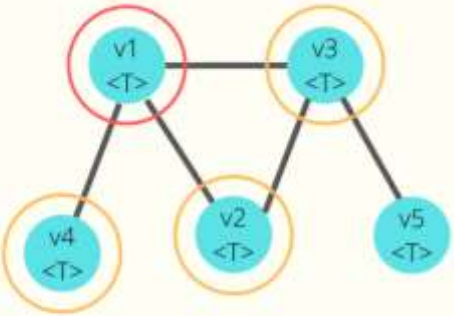
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges-1}}



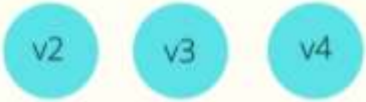
getNeighborts(graph, vertex)

“Returns a collection of vertices that it are neighbor to vertex indicated”

{pre: graph ={Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges} ∧ vertex ∈ graph }



{post: = {v1, v2, .. vn } n <= Vertices ∧ ∀i / 1<= i <= Vertices → {vi, vertex } ∈ Edges of graph }



getNumberOfVertices(graph)

“Returns an integer represents the Vertices value”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }

{post: <vertices> }

getNumberOfEdge(graph)

“Returns an integer represents the Edge value”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }

{post: <edges> }

areAdjacent(graph, vertex1, vertex2)

“Verify if vertex1 and vertex2 area adjacent”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph }

{post: TRUE if {vertex1, vertex2} ∈ Edges of graph on the contrary FALSE}

isInGraph(graph, object)

“Verify if object is in graph”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } ⋀ object ∈ T }

{post: TRUE if object is in any vertex of graph on the contrary FALSE }

getEdgeWeight(graph, vertex1, vertex2)

“Returns the edge weight of graph”

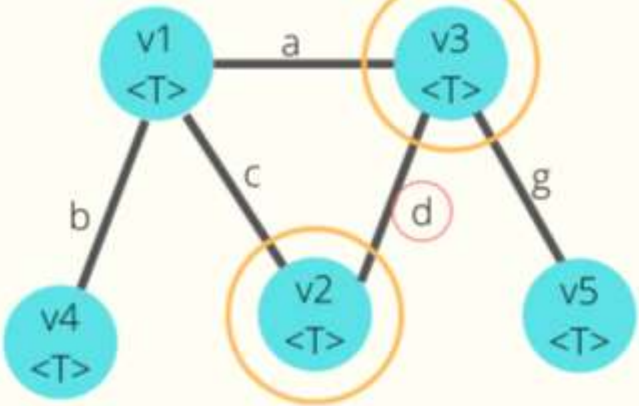
{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph }

{post: weight of {vertex1, vertex2} }

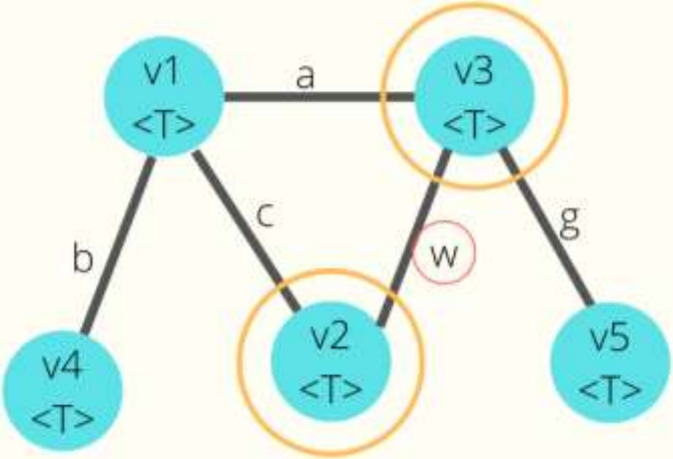
setEdgeWeight(graph, vertex1, vertex2, w)

“Returns the edge weight of graph”

{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph ⋀ w ∈ double}



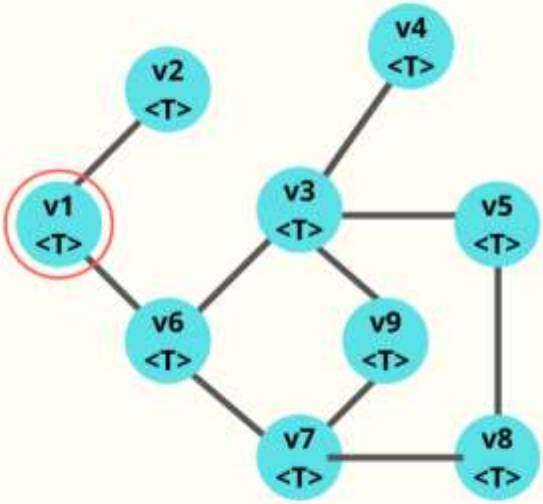
{post: weight of {vertex1, vertex2} = w}



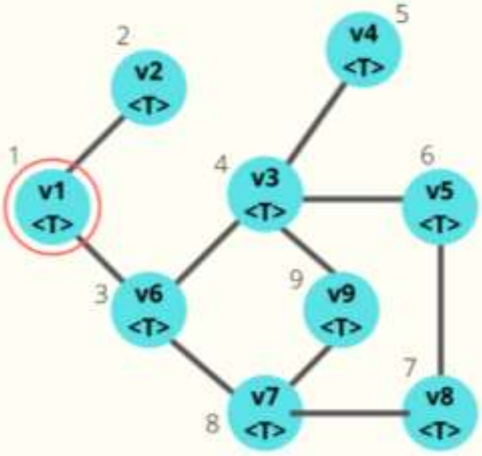
dfs(graph, vertex)

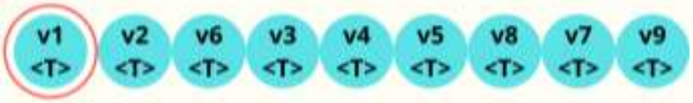
“Returns an ordered collection of vertices that represents the deep path (Depth First Search) of the graph starting at vertex”

{pre: graph = {Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges}
⋀ vertex ∈ graph ⋀ graph is united}



{post: = {v1, v2, .. vn } n <= Vertices }

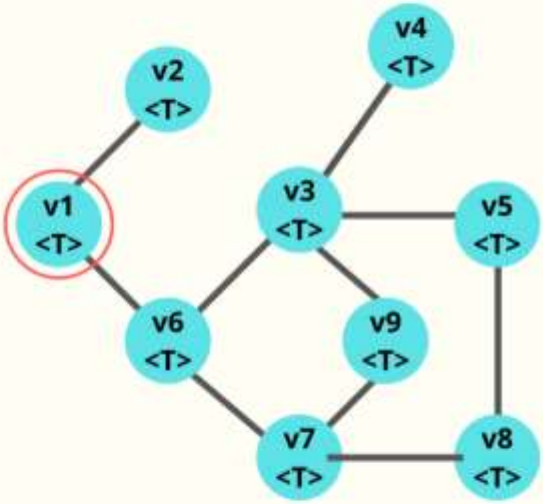




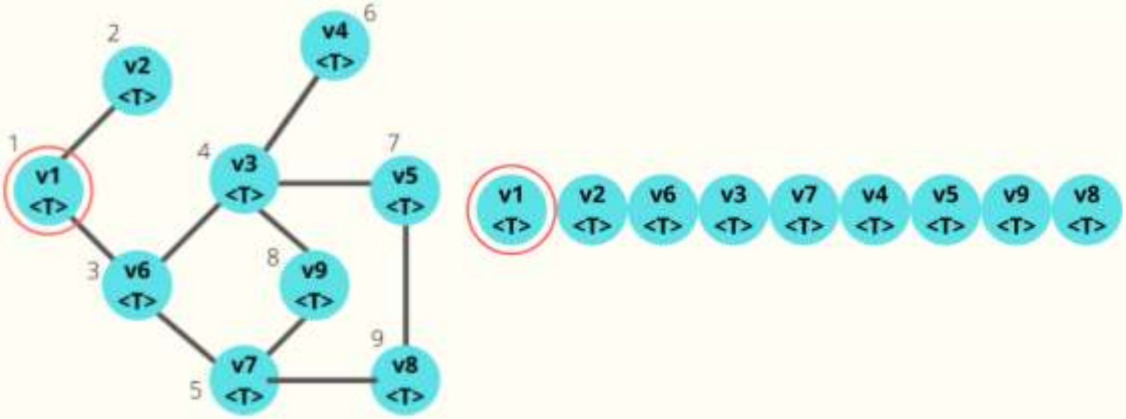
bfs(graph, vertex)

“Returns an ordered collection of vertices that represents the amplitude path (Breadth First Search) of the graph starting at vertex”

{pre: graph = {Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges}
⋀ vertex ∈ graph ⋀ graph is united}



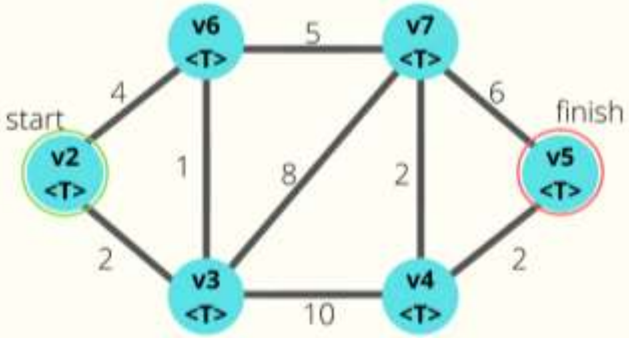
{post: = {v1, v2, .. vn } n <= Vertices }



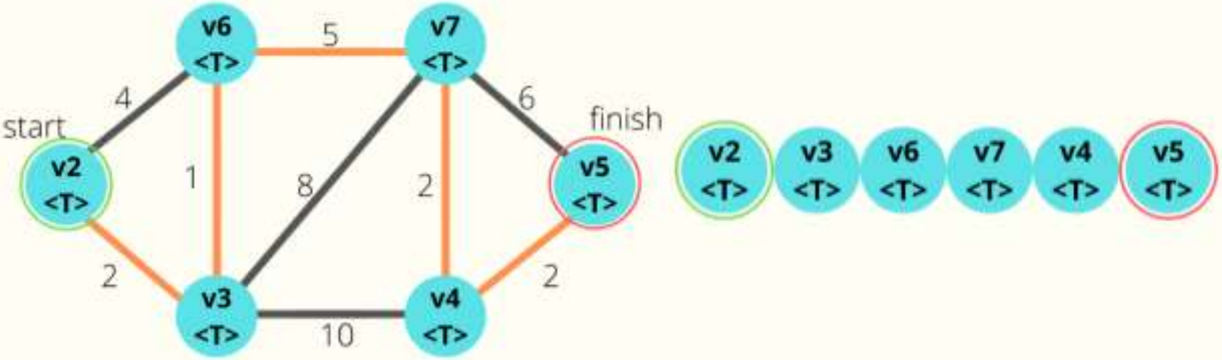
dijkstra(graph, vertex1, vertex2)

“Returns the path of least weight between vertex1 and vertex2”

{pre: graph = {Directed = directed, Weighted = TRUE, Vertices = vertices, Edge = edges}
⋀ vertex1, vertex2 ∈ graph ⋀ graph is united}



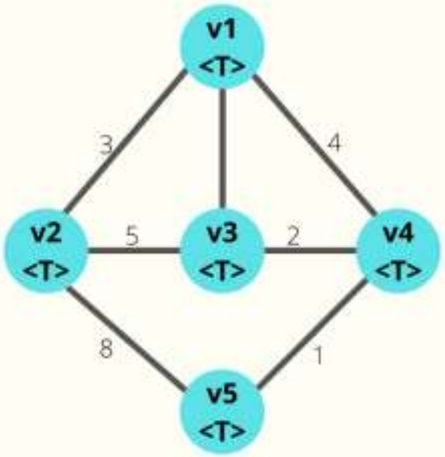
{post: g= {v1, v2, .. vn } n <= Vertices }

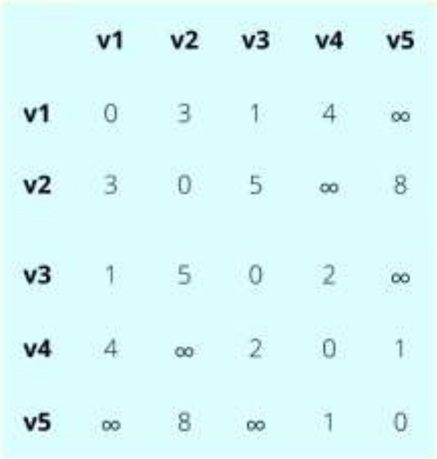


floydWarshall(graph)

“Returns a matrix with the lowest weight between all the vertices”

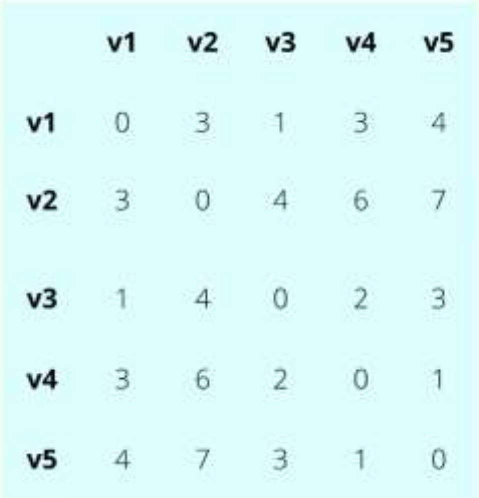
{pre: graph = {Directed = directed, Weighted = TRUE, Vertices = vertices, Edge = edges} Λ graph is united}





	v1	v2	v3	v4	v5
v1	0	3	1	4	∞
v2	3	0	5	∞	8
v3	1	5	0	2	∞
v4	4	∞	2	0	1
v5	∞	8	∞	1	0

{post: the matrix with the lowest weight between all the vertices of graph }

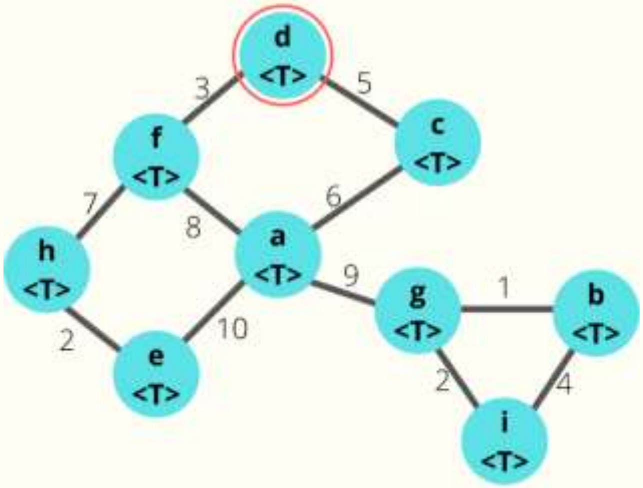


	v1	v2	v3	v4	v5
v1	0	3	1	3	4
v2	3	0	4	6	7
v3	1	4	0	2	3
v4	3	6	2	0	1
v5	4	7	3	1	0

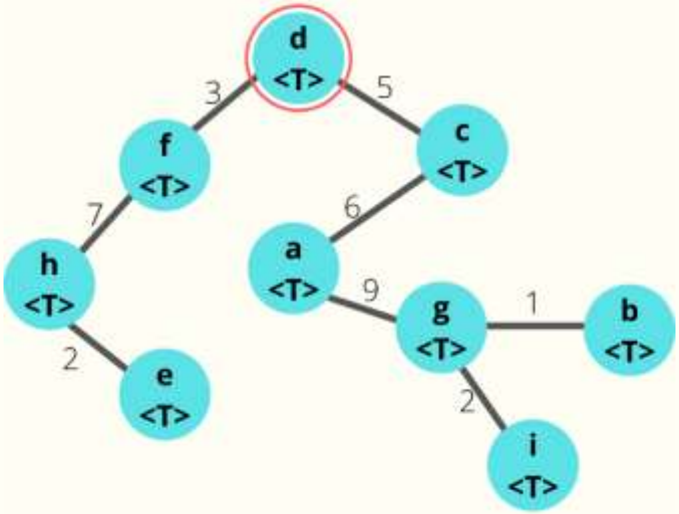
prim(graph, vertex)

"Returns the minimum spanning tree (MST) of graph, whit root in vertex"

{pre: graph = {Directed = directed, Weighted = TRUE, Vertices = vertices, Edge = edges}
⋀ vertex ∈ graph ⋀ graph is united}



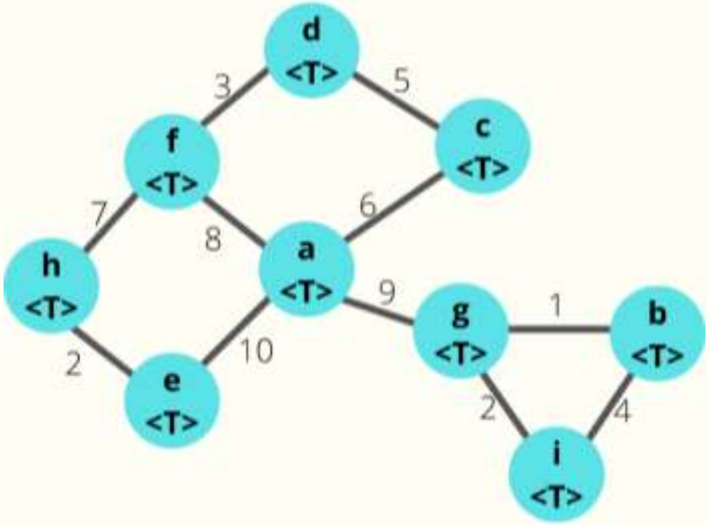
{post: new Graph = {Directed = TRUE, Weighted = TRUE, Vertices = vertices, Edge = vertices-1} }



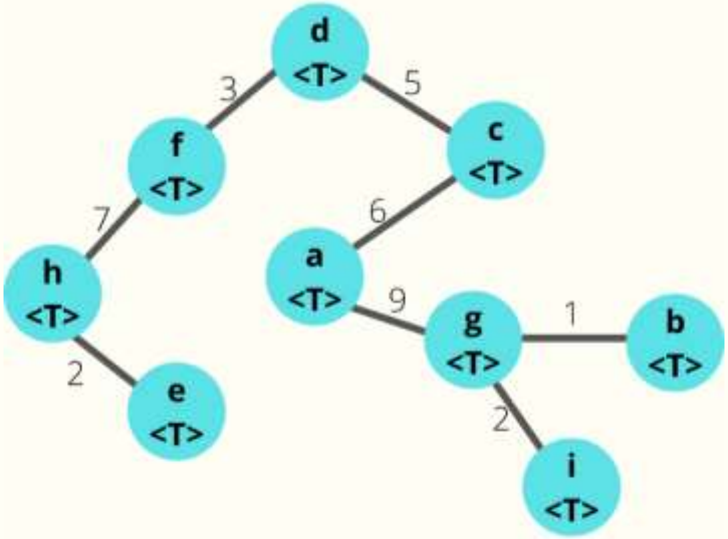
kruskal(graph)

“Returns the minimum spanning tree (MST) of graph”

{pre: graph = {Directed = directed, Weighted = TRUE, Vertices = vertices, Edge = edges} Λ graph is united}



{post: new Graph = {Directed = TRUE, Weighted = TRUE, Vertices = vertices, Edge = vertices-1} }



searchVertex(graph, object)

“Returns the vertex that contains the object in the graph”

{pre: graph = {Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges} ⋀ object ∈ T }

{post: vertex = {.., Value = object, … } ∈ graph if it isn’t in the graph returns null }

getEdges(graph)

“Returns a collection with the edges of the graph”

{pre: graph = {Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges} }

{post: {E1, E2, E3, …, En } n = Edges ⋀ ∀i ∀j / 1<= I, j <= vertices → {vi, vj} ∈ Edges of graph } }

getContests(graph)

“Returns a collection with the T type elements of the graph that represents the vertex”

{pre: graph = {Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges} }

{post: {E1, E2, …, En } n = Vertices ⋀ ∀i / 1<= i<= Edges → ∃j / 1<= j<= Vertices vj = {.., Value = Ei, … }
⋀ vj ∈ Vertex<T> ⋀ vj ∈ graph } }