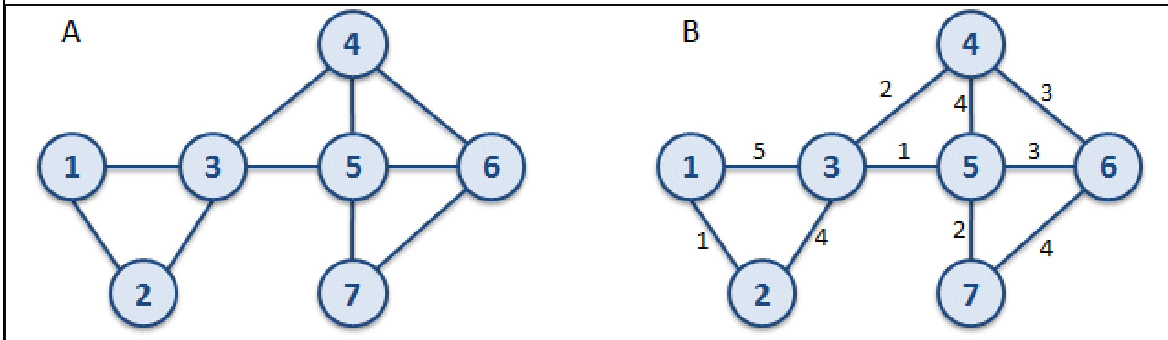
TAD Graph

Graph = $\{V = \{v_1, v_2, ..., v_n\}, E = \{e_1 = (v_{i1}, v_{j1}, w_1), e_2 = (v_{i2}, v_{j2}, w_2), e_m = (v_{im}, v_{jm}, w_m)\},$ directed, weighted$\}$



{inv:

1. $\forall e_k \in E, v_{ik} \in V \wedge vj_k \in V, w_k > 0$
2. directed = false $\Rightarrow (\forall(a,b) \in E \; \exists \; (b,a) \in E, a, b \in V)$
3. weighted = false $\Rightarrow \forall e_k \in E, w_k = 1$

}

Primitive Operations

• Graph	<> → <Graph> Constructor
• addVertex	<Vertex> → <Graph> Modifier
• addEdge	<Vertex, Vertex> → <Graph> Modifier
• addEdge	<Vertex, Vertex, double> → <Graph> Modifier
• removeVertex	< Vertex> → <Graph> Modifier
• removeEdge	< Vertex, Vertex> → <Graph> Modifier
• getNumberOfVertices	<> → <Integer> Analyzer
• getNumberOfEdges	<> →< Integer> Analyzer
• areAdjacent	< Vertex, Vertex >→ <boolean> Analyzer
• searchInGraph	< T >→ <boolean> Analyzer
• getVertex	<Graph> → List<Vertex> Analyzer
• isDirected	<> → <boolean> Analyzer
• isWeighted	<> → <boolean> Analyzer
• bfs	<Vertex> → <Graph> Analyzer
• dfs	<> → <Graph> Analyzer
• dijkstra	< Vertex> → <Graph> Analyzer
• floyd-warshall	<> → <Double[][]>Analyzer
• prim	< Vertex >→ <Graph> Analyzer
• kruskal	<> → <Graph> Analyzer

Operations

Graph (boolean directed, boolean weighted, int n)
Create a new graph that may or may not be directed or weighted.
{pre: }
{post: Graph = {V={}, E={}, directed, weighted }

addVertex (Vertex v)
Insert a vertex in the graph.
{pre: v ∉ g.V}
{post: v ∈ g.V}

addEdge (Vertex v1, Vertex v2)
Add an edge of weight 1 that goes from v1 to v2. If the graph is not directed, it also adds it from v2 to v1.
{pre: v1, v2 ∈ g.V }
{post: edge = (v1, v2, 1) ∈ g.E. If g.directed = false, edge = (v2, v1, 1)) ∈ g.E }

addEdge (Vertex v1, Vertex v2, double weight)
Add an edge of weight 1 that goes from v1 to v2. If the graph is not directed, it also adds it from v2 to v1.
{pre: v1, v2 ∈ g.V, g.weight = true, w > 0}
{post: edge = (v1, v2, weight) ∈ g.E. If g.directed = false, edge = (v2, v1, weight)) ∈ g.E }

removeVertex (Vertex v)
Eliminate v from the graph
{pre: v ∈ g.V }
{post: v ∉ g.V. All vertices that are incidents with v ∉ g. E }

removeEdge (Vertex v1, Vertex v2)
Eliminate the edge that goes from v1 to v2 in the graph
{pre: v1, v2 ∈ g.V, (v1,v2,w) ∈ g.E }
{post: edge= (v1,v2,w) ∉ g.E. If g.directed = false, e’ = (v2,v1, w)) ∉ g.E }

getNumVertex ()
Returns the number of vertices in the graph
{pre: }
{post: number of vertices}

getNumEdges ()
Returns the number of edges in the graph.
{pre:}
{post: number of edges}

searchInGraph (T value)
Returns if there is a vertex with the given value in the graph.
{pre: }
{post: true if $\exists x \in g.V$: value

idDirected ()
Returns if the graph is directed
{pre: }
{post: true if is directed, false otherwise}

isWeighted()
Returns if the graph is weighted.
{pre:}
{post: true if is directed, false otherwise}

areAdjacent (Vertex v1, Vertex v2)
Returns if there is an edge from x to y
{pre: v1, v2 ∈ g.V }
{post: true if (v1, v2, w) ∈ g.E.}

dijkstra (Vertex v)
Carry out the Dijkstra algorithm, taking v as the initial vertex
{pre: v ∈ g.V, g }
{post: ∀v ∈ g.V, adds attributes v.pred and v.d, corresponding respectively to the predecessor and the distance added by Dijkstra's algorithm}

bfs(Vertex v)
Performs the Breadth First Search algorithm, adjusting information for the vertices of the graph.
{pre v ∈ g.V, g:}
{post: ∀v ∈ g.V, adds attributes v.pred and v.d, which correspond to those added by the Breadth First Search algorithm}

dfs ()
Performs the Depth First Search algorithm, adjusting information for the vertices of the graph
{pre: }
{post: ∀v ∈ g.V, adds attributes v.pred, v.d and v.f, which correspond to those added by the Depth First Search algorithm}

floyd-warshall ()
Perform the Floyd-Warshall algorithm on graph.
{pre: }
{post: Returns the dist matrix, where position [i, j] represents the minimum distance to go from vertex vi to vj }

kruskal ()
Perform Kruskal's algorithm on the graph.
{pre: }
{post: {e1, e2,…, en}, where ei ∈ g.E are the edges that belong to the MST formed by Kruskal }

prime (Vertex v)
Perform Prim's algorithm taking r as the root of the tree, adjusting information for the vertices of the graph.
{pre: v ∈ g.V, g is not directed }
{post: ∀v ∈ g.V, adds attributes v.pred and v.d, which correspond respectively to the predecessor and the key added by Prim's algorithm}