Algorithms and data structures

Final Proyect - 2019-2

Daniel Villota - A00356255, Natalia Gonzalez - A00354849

Engineering Method

(problem 1 – 1112 Mice and Maze)

Problem definition:

Judge: Online Judge
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3553

link: https://drive.google.com/file/d/12zpE-cB_zDxe5ajTpYpyCBEaEufSy3M9/view?usp=sharing

Phase 1: Problem identification

The question that defines this problem is: How many mice manage to leave the maze in time?

Phase 2: Collection of the necessary information

All the information necessary to understand the problem is provided by the statement, the entries are specific and it is the only thing that will be obtained.

Phase 3: Search for creative solutions

To solve this problem, an algorithm is needed to find the path of the lowest weight of the labyrinth, from each of the cells. For this you can use algorithms known as:

- Floyd-Warshall
- Dijkstra

Another option may be to look for all possible paths until one meets the time limit and if it is not possible to discard it

Phase 4: Transition of the formulation of ideas to preliminary designs

The idea of looking for all possible paths is likely to function, but the complexity is the highest compared to the other options, therefore it is ruled out.

On the other hand, the Floyd Warshall option has a complexity of V^3 and the minimum weight of all roads is obtained

Also, the Dijkstra algorithm has a temporal complexity of V to 2 with a single path, so this algorithm must be repeated V times, therefore, the serious complexity of V^3

UNIVERSIDAD ICESI

Phase 5: Evaluation and selection of the best solution

To solve this problem, Floyd Warshall was selected, since it is easier to implement and at the end of the algorithm the answer is already obtained, while for the Dijkstra algorithm, it is necessary to repeat the algorithm to verify if each mouse manages to exit

Phase 6: Preparation of reports and specifications

Functional requirements

Name	**R.# 1. Find the number of mice that manage to get out of the maze**
Summary	Find the number of mice that leave the maze in the given time
Input	
• Number of cages • Departure location • Maximum time determined to exit • Connection between cages • Time it takes to go from one cage to another Results	
Output	
A number that indicates the number of mice that manage to get out of the maze	

Name	**R.# 2. Visualize the maze**
Summary	Summary Show the shape of the maze
Inputs	
• Number of cages • Departure location • Maximum time determined to exit • Connection between cages • Time it takes to go from one cage to another	
Output	
A screen maze display	

Nombre	**R.# 3. Visualize the cages of the mice that manage to leave the labyrinth**
Resumen	It allows to visualize the mouse cages that manage to leave the labyrinth
Input	
None	
Output	
A visualization of the maze on the screen with the cages of the mice that managed to leave a different color	

UVa Online Judge

1112 Mice and Maze

A set of laboratory mice is being trained to escape a maze. The maze is made up of cells, and each cell is connected to some other cells. However, there are obstacles in the passage between cells and therefore there is a time penalty to overcome the passage Also, some passages allow mice to go one-way, but not the other way round.

Suppose that all mice are now trained and, when placed in an arbitrary cell in the maze, take a path that leads them to the exit cell in minimum time.

We are going to conduct the following experiment: a mouse is placed in each cell of the maze and a count-down timer is started. When the timer stops we count the number of mice out of the maze.

Write a program that, given a description of the maze and the time limit, predicts the number of mice that will exit the maze. Assume that there are no bottlenecks is the maze, i.e. that all cells have room for an arbitrary number of mice.

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The maze cells are numbered $1, 2, \ldots, N$, where N is the total number of cells. You can assume that $N \leq 100$.

The first three input lines contain N, the number of cells in the maze, E, the number of the exit cell, and the starting value T for the count-down timer (in some arbitrary time unit).

The fourth line contains the number M of connections in the maze, and is followed by M lines, each specifying a connection with three integer numbers: two cell numbers a and b (in the range $1, \ldots, N$) and the number of time units it takes to travel from a to b.

Notice that each connection is one-way, i.e., the mice can't travel from b to a unless there is another line specifying that passage. Notice also that the time required to travel in each direction might be different.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

The output consists of a single line with the number of mice that reached the exit cell E in at most T time units.

Sample Input

```
1

4
2
1
8
1 2 1
1 3 1
```

```
2 1 1
2 4 1
3 1 1
3 4 1
4 2 1
4 3 1
```

Sample Output

```
3
```

Engineering Method

(problem 2 – 11631 Dark roads)

Problem definition:

Judge: Online
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2678

link: https://drive.google.com/file/d/1gN7QIJUoG5PKY4rR4p72jPhRXBAQ62Ch/view?usp=sharing

Phase 1: Problem identification

The main problem is knowing the maximum amount of money that the government can save without sacrificing the security of citizens

Phase 2: Collection of the necessary information

All the information necessary to understand the problem is provided by the statement, the entries are specific and it is the only thing that will be obtained.

Phase 3: Search for creative solutions

To solve this problem, we must implement an algorithm that allows us to simulate the map of the city and find the roads that have the least cost and it is possible to reach all places. For this you can use algorithms that already exist as Prim or Kruskal.

Phase 4: Transition of the formulation of ideas to preliminary designs

The Prim algorithm has a temporal complexity of O (n2) and the Kruskal algorithm has a similar complexity of O (n * log (n)).

Phase 5: Evaluation and selection of the best solution

For ease of implementation it was decided to use the Prim algorithm.

UNIVERSIDAD ICESI

Phase 6: Preparation of reports and specifications

Functional requirements

Name	**R.# 1. Find the maximum amount of money that can be saved**
Summary	Find the maximum amount of money that can be saved without sacrificing the safety of citizens
Input	
• Number of places • Number of roads • Connection between places • The distance of the connections	
Output	
A number that indicates the money that can be saved with this method	

Name	**R. # 2. Display the shape of the city**
Summary	Display the shape of the city
Input	
• Number of places • Number of roads • Connection between places • The distance of the connections	
Output	
A display of the city by screen	

Name	**R.# 3. View the route that is on and connect all places in the city**
Summary	View the route that remains on
Input	
None	
Output	
A visualization of the shortest streets that connect all places in the city.	

UVa Online Judge

11631 Dark roads

Economic times these days are tough, even in Byteland. To reduce the operating costs, the government of Byteland has decided to optimize the road lighting. Till now every road was illuminated all night long, which costs 1 Bytelandian Dollar per meter and day. To save money, they decided to no longer illuminate every road, but to switch off the road lighting of some streets. To make sure that the inhabitants of Byteland still feel safe, they want to optimize the lighting in such a way, that after darkening some streets at night, there will still be at least one illuminated path from every junction in Byteland to every other junction.

What is the maximum daily amount of money the government of Byteland can save, without making their inhabitants feel unsafe?

Input

The input file contains several test cases. Each test case starts with two numbers m and n, the number of junctions in Byteland and the number of roads in Byteland, respectively. Input is terminated by $m = n = 0$. Otherwise, $1 \leq m \leq 200000$ and $m - 1 \leq n \leq 200000$. Then follow n integer triples x, y, z specifying that there will be a bidirectional road between x and y with length z meters ($0 \leq x, y < m$ and $x \neq y$). The graph specified by each test case is connected. The total length of all roads in each test case is less than 2^{31}.

Output

For each test case print one line containing the maximum daily amount the government can save.

Sample Input

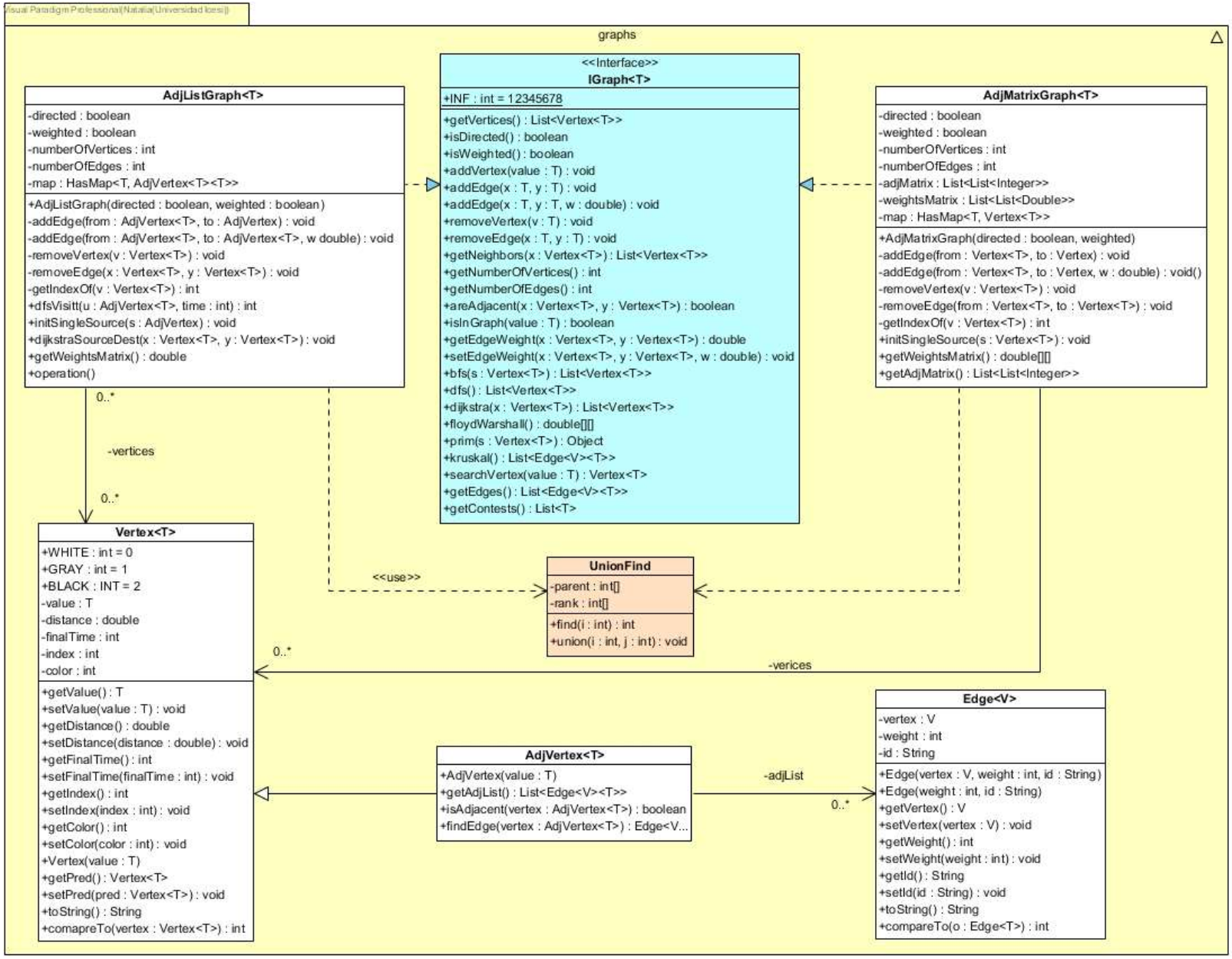
```
7 11
0 1 7
0 3 5
1 2 8
1 3 9
1 4 7
2 4 5
3 4 15
3 5 6
4 5 8
4 6 9
5 6 11
0 0
```

Sample Output

```
51
```

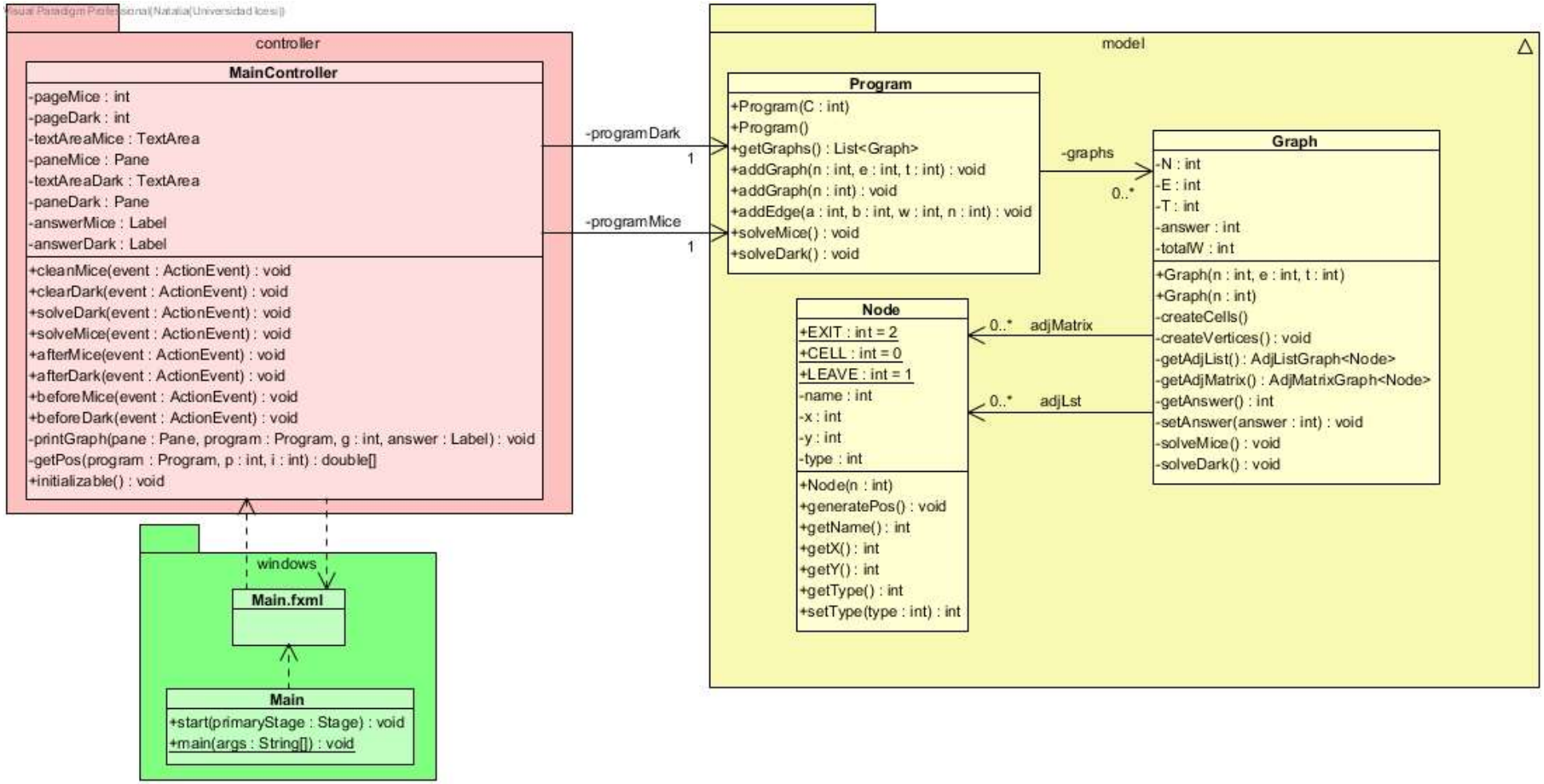
UNIVERSIDAD ICESI

Class Diagram Graph Implementation



UNIVERSIDAD ICESI

Class Diagram for Problems



Jesús Daniel Villota
Natalia González

UNIT TESTING

Test 1: Verify if the method addVertex adds a vertex to the graph correctly.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	addVertex	An empty graph	One single vertex	The graph now contains one vertex.
Graph	addVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4	One vertex with a value of 5	The graph now contains five vertices and the vertex most recently added.
Graph	addVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4	One vertex with a value of 1	The graph still contains four vertices and the vertex with value of 1 was not added because another vertex with the same value already exists.

Test 2: Verify if the method addEdge adds a weighted - directed edge to the graph correctly.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	addEdge	A graph that contains four vertices with the next values: 1, 2, 3, 4	Vertex x = 1 Vertex y = 4 Weight = 5	The graph now contains an edge between the vertices 1 and 4 with a weight of 5.
Graph	addEdge	A graph that contains four vertices with the next values: 1, 2, 3, 4 And contains an edge from	Vertex x = 1 Vertex y = 4 Weight = 5	The graph already contains an edge from 1 to 4 with a weight of 5, so there are no changes

		1 to 4 with a weight of 5.		made in the graph.
Graph	addEdge	A graph that contains four vertices with the next values: 1, 2, 3, 4	Vertex x = 2 Vertex y = 2 Weight = 10	An edge from 2 to 2 (a loop) with a weight of 10 was added to the graph.
Graph	addEdge	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 4, 5]	Vertex x = 2 Vertex y = 3 Weight = 5	An edge from 2 to 3 was added. Now vertices 4 and 3 are adjacent to vertex 2.

Test 3: Verify if the method removeVertex removes the specified vertex from the graph and every connection to it correctly.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	removeVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 4, 5]	Vertex x = 2	The graph now contains 3 vertices: 2,3,4 And the next edge: x, y, w [3, 4, 5]
Graph	removeVertex	A graph that contains three vertices with the next values: 2, 3, 4	Vertex x = 1	The graph still contains three vertices with the next values: 2,3,4

		And the next edge: x, y, w [3, 4, 5]		And the next edge: x, y, w [3, 4, 5] No changes done.
Graph	removeVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 3, 3] [1, 4, 6] [1, 3, 5]	Vertex x = 2	The graph now contains three vertices with the next values: 1,3,4 And the next edges: x, y, w [1, 3, 3] [1, 4, 6] [1, 3, 5]
Graph	removeVertex	An empty graph	Vertex x = 1	The graph is still empty. There aren't edges to remove.

Test 4: Verify if the method removeEdge removes the specified edge from the graph.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	removeEdge	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]	Vertex x = 2 Vertex y = 4	The graph still has the same four vertices: 1, 2, 3, 4 And the remaining edges are: x, y, w [1, 2, 3] [4, 2, 3] [3, 1, 5]
Graph	removeEdge	A graph that contains four vertices with	Vertex x = 1 Vertex y = 3	The graph still has the same four vertices:

		the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]		1, 2, 3, 4 And no edge was removed because the specified edge doesn't exist in the actual graph. The remaining edges are: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]
Graph	removeEdge	An empty graph	Vertex x = 1 Vertex y = 2	The graph is still empty and there aren't edges to remove.

Test 5: Verify that the method getVertex returns the vertex correspondent to the given value if and only if the graph contains the vertex.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	getVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]	Value x = 2	The method returns the vertex correspondent to the value of 2.
Graph	getVertex	A graph that contains four vertices with the next values: 1, 2, 3, 4	Value x = 5	The method returns a NIL value because the specified value doesn't have any related vertex

		And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]		in the actual graph.
Graph	getVertex	An empty graph	Value x = 1	The method returns a NIL value because it's empty.

Test 6: Verify that the method areAdjacent returns the correct Boolean value according to the correspondent given vertices and if the correct connection exists between them.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	areAdjacent	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]	Vertex x = 2 Vertex y = 4	The method returns true, because there is an edge from vertex 2 to vertex 4 in the actual graph.
Graph	areAdjacent	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 2, 3] [3, 1, 5]	Vertex x = 1 Vertex y = 4	The method returns false, because there is not an edge from vertex 1 to vertex 4 in the actual graph.

Graph	areAdjacent	An empty graph	Vertex x = 1 Vertex y = 2	The method returns false because the given vertices don't even exist in the actual graph.

Test 7: Verify that the method bfs adjusts the information of the vertices in the given graph correctly, according to what the known algorithm is supposed to do.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	bfs	A graph that contains four vertices with the next values: 1, 2, 3, 4, 5 And the next edges: x, y, w [1, 2, 3] [2, 4, 6] [4, 3, 3] [1, 3, 5] [4, 5, 2]	Vertex source = 1	The vertices in the actual graph now have the information assigned as it follows: 1.pred = NIL 2.pred = 1 3.pred = 1 4.pred = 2 5.pred = 4 1.dist = 0 2.dist = 3 3.dist = 5 4.dist = 9 5.dist = 11

Test 8: Verify that the method dfs adjusts the information of the vertices in the given graph correctly, according to what the known algorithm is supposed to do.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	dfs	A graph that contains four vertices with the next values: 1, 2, 3, 4, 5 And the next edges:	Vertex source = 4	The vertices in the actual graph now have the information assigned as it follows: 1.pred = NIL

		x, y, w [1, 2, 3] [2, 4, 6] [4, 3, 3] [1, 3, 5] [4, 5, 2]		2.pred = NIL 3.pred = 4 4.pred = NIL 5.pred = 4 1.dist = INF 2.dist = INF 3.dist = 3 4.dist = 0 5.dist = 2

Test 9: Verify that the method prim assigns the correct values to the vertices so that it corresponds to forming a minimum spanning tree.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	prim	A graph that contains four vertices with the next values: 1, 2, 3, 4, 5 And the next edges: x, y, w [1, 2, 2] [2, 1, 2] [1, 3, 12] [3, 1, 12] [2, 3, 7] [3, 2, 7] [2, 4, 15] [4, 2, 15] [3, 4, 3] [4, 3, 3] [2, 5, 4] [5, 2, 4] [4, 5, 6] [5, 4, 6]	None	The vertices in the actual graph now have the information assigned as it follows: 1.pred = NIL 2.pred = 1 3.pred = 4 4.pred = 5 5.pred = 2 1.dist = 0 2.dist = 2 3.dist = 15 4.dist = 12 5.dist = 6

Test 10: Verify that the method kruskal assigns the correct values to the vertices so that it corresponds to forming a minimum spanning tree.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	kruskal	A graph that contains four	None	The method returns a list

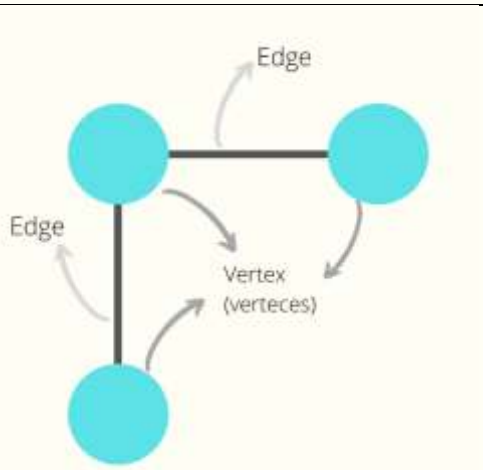
		vertices with the next values: 1, 2, 3, 4, 5 And the next edges: x, y, w [1, 2, 2] [2, 1, 2] [1, 3, 12] [3, 1, 12] [2, 3, 7] [3, 2, 7] [2, 4, 15] [4, 2, 15] [3, 4, 3] [4, 3, 3] [2, 5, 4] [5, 2, 4] [4, 5, 6] [5, 4, 6]		with the edges that conforms the minimum spanning tree from the actual graph which are the following ones: { (1,2), (3,4), (2,5), (4,5) }

Test 11: Verify that the method Dijkstra assigns the correct values to the vertices so that it matches with the shortest path between the given vertices.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	dijkstra	A graph that contains four vertices with the next values: 1, 2, 3, 4, 5 And the next edges: x, y, w [1, 2, 5] [1, 3, 1] [1, 4, 3] [2, 5, 8] [4, 2, 2] [4, 5, 5] [4, 3, 2]	Vertex source = 1 Vertex dest = 5	The vertices in the actual graph now have the information assigned as it follows: 1.pred = NIL 2.pred = 4 3.pred = 1 4.pred = 1 5.pred = 2 1.dist = 0 2.dist = 5 3.dist = 1 4.dist = 3 5.dist = 10

Test 12: Verify that the method Floyd Warshall finds the correct minimum distances from every vertex to any other vertex in the graph.				
Class	**Method**	**Scenario**	**Input**	**Output**
Graph	floydWarshall	A graph that contains four vertices with the next values: 1, 2, 3, 4 And the next edges: x, y, w [1, 2, 1] [2, 3, 2] [1, 4, 5] [3, 4, 1]	None	The method returns the next matrix of distances: 1 2 3 4 1 [0 1 3 4] 2 [∞ 0 2 3] 3 [∞ ∞ 0 1] 4 [∞ ∞ ∞ 0]

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"

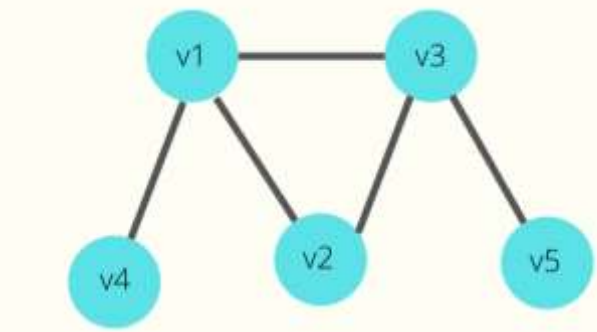
{ pre: TRUE ∧ directed ∈ Boolean ∧ weighted ∈ Boolean}

{ 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}

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"

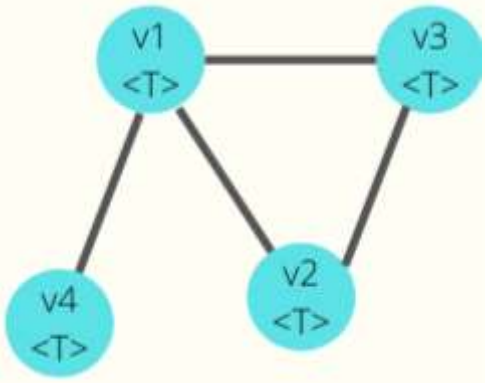
{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges } }

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

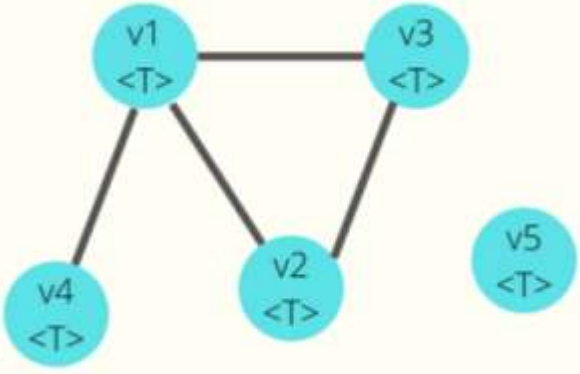
addVertex(graph, object)

"Add a new vertex to graph"

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



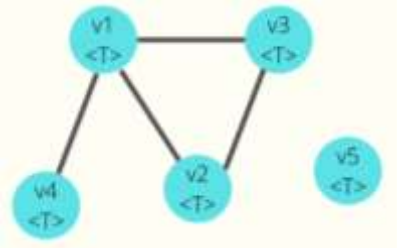
{post: graph ={ Directed = directed, Weighted = weighted, Vertices = vertices+1, Edge = edges } }



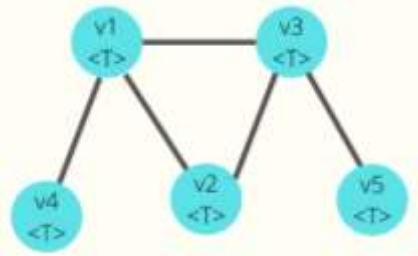
addEdge(graph, vertex1, vertex2)

"Add a new edge between two vertex of graph"

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



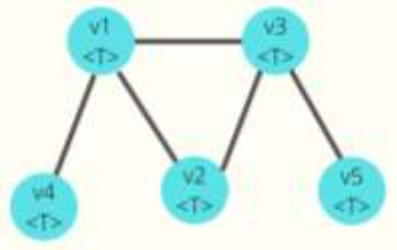
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges +1}}



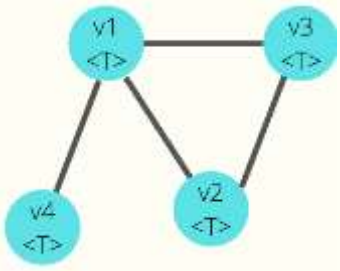
removeVertex(graph, vertex1)

"Remove a vertex of the graph"

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



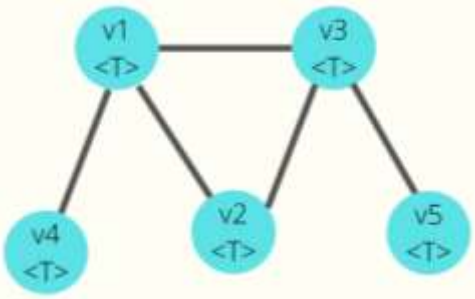
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices-1, Edge <=edges}}



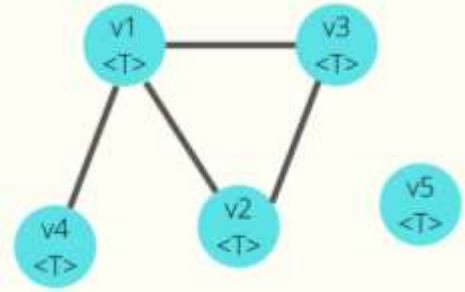
removeEdge(graph, vertex1, vertex2)

"Remove a connection between two vertices of graph"

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



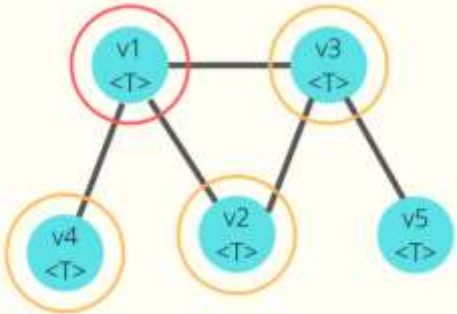
{post: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges-1}}



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"

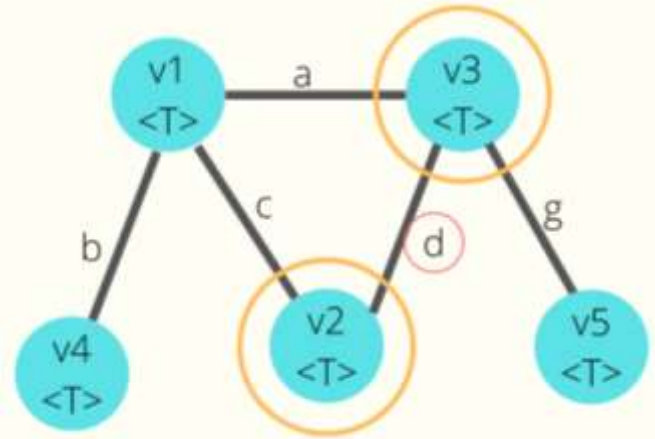
{pre: graph = { Directed = directed, Weighted = weighted, Vertices = vertices, Edge = edges }
⋀ vertex1 ∈ graph ⋀ vertex2 ∈ graph }

{post: weight of {vertex1, vertex2} }

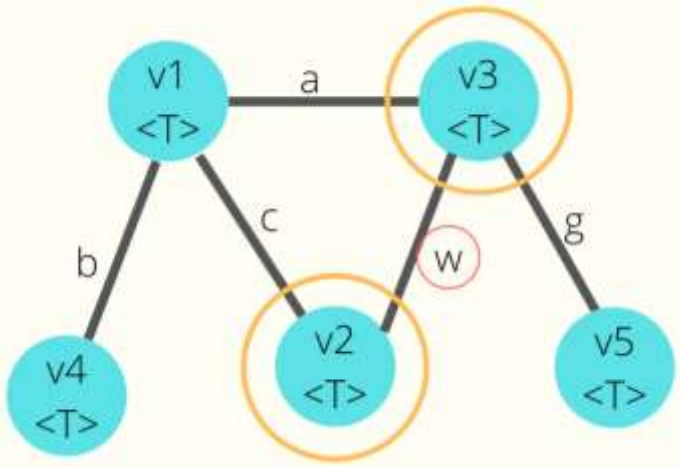
setEdgeWeight(graph, vertex1, vertex2, w)

"Returns the edge weight of graph"

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



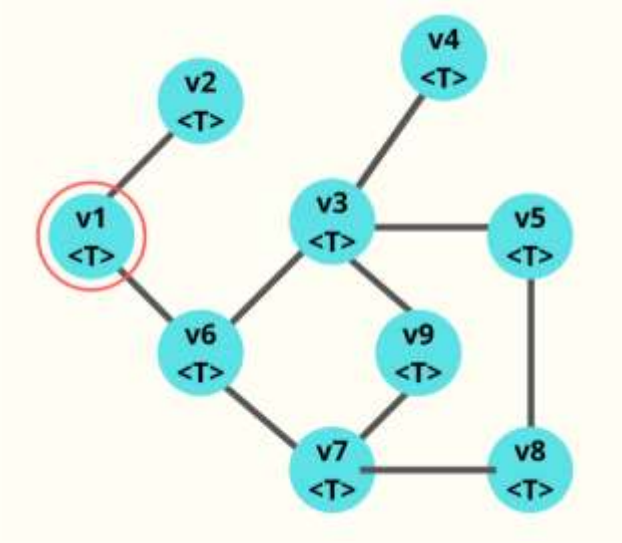
{post: weight of {vertex1, vertex2} = w}



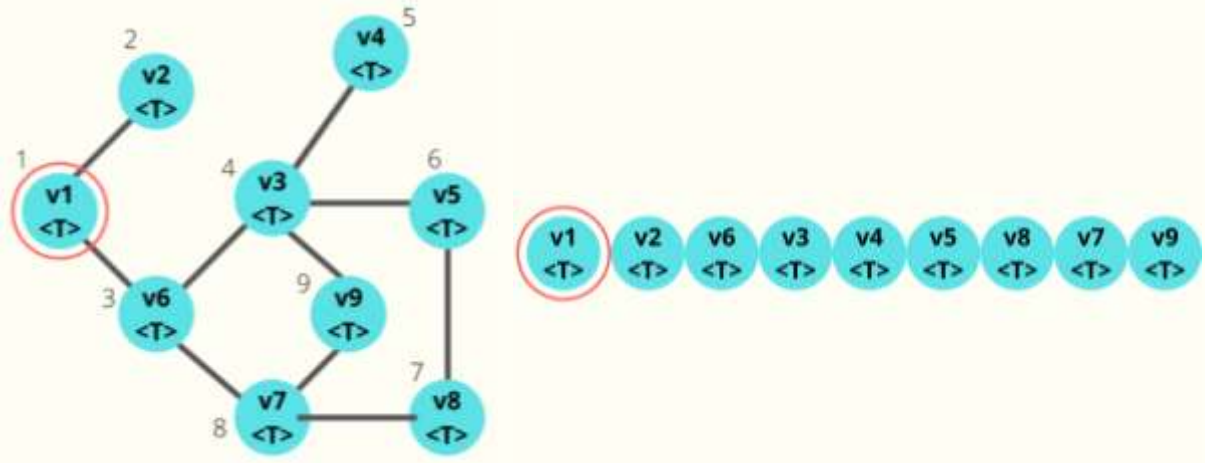
dfs(graph, vertex)

"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}
Λ vertex ∈ graph Λ graph is united}



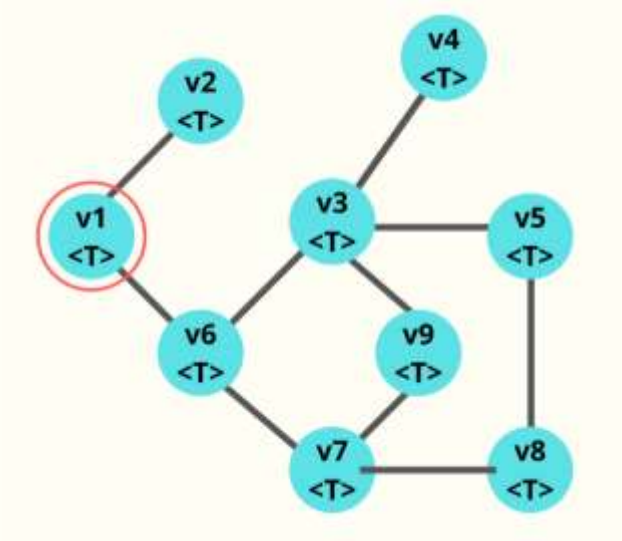
{post: = {v1, v2, .. vn } n <= Vertices }



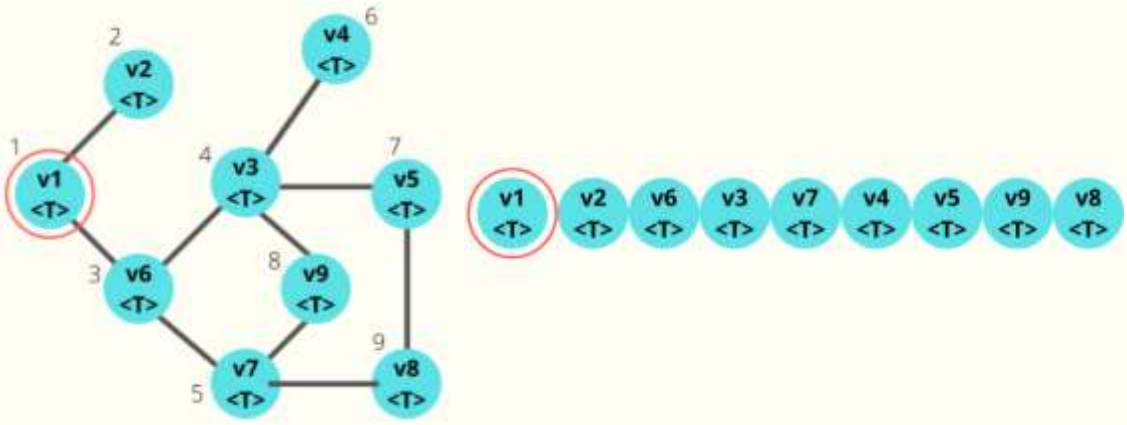
bfs(graph, vertex)

"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}
Λ vertex ∈ graph Λ graph is united}



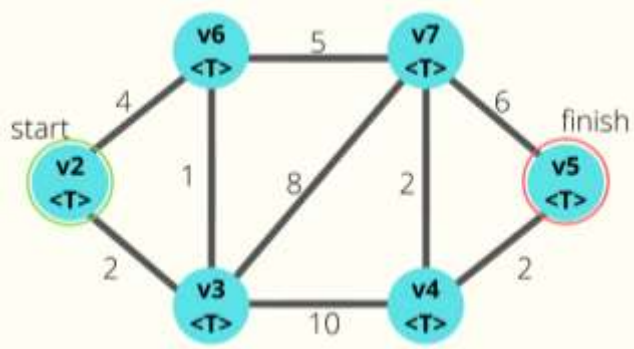
{post: = {v1, v2, .. vn } n <= Vertices }



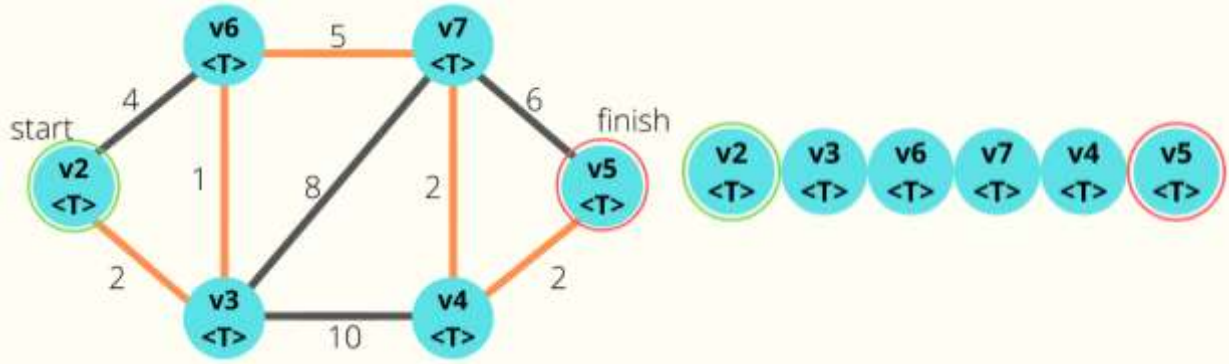
dijkstra(graph, vertex1, vertex2)

"Returns the path of least weight between vertex1 and vertex2"

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



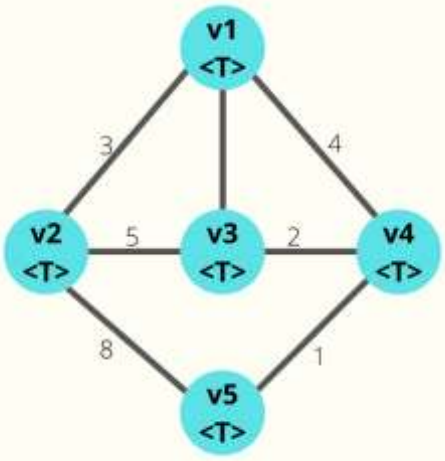
{post: g= {v1, v2, .. vn } n <= Vertices }



floydWarshall(graph)

"Returns a matrix with the lowest weight between all the vertices"

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

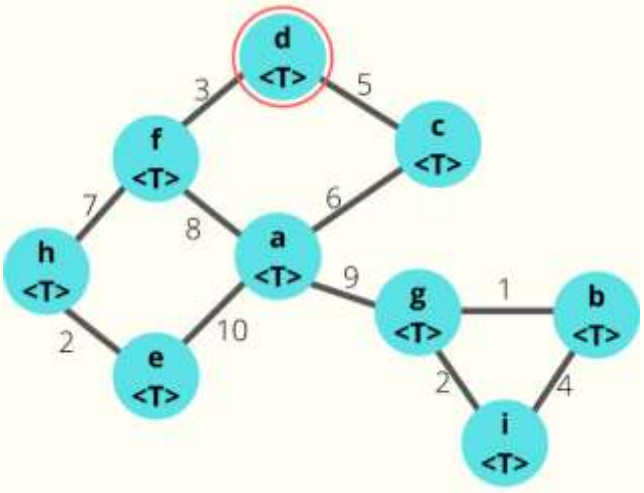
{post: the matrix with the lowest weight between all the vertices of graph }

	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

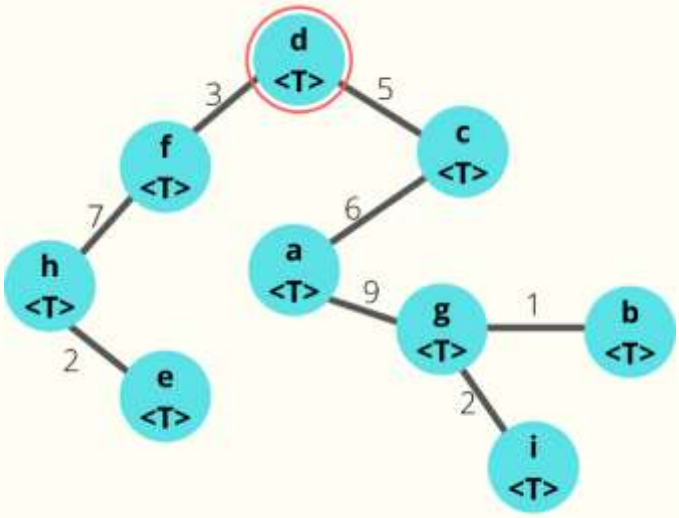
prim(graph, vertex)

"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}



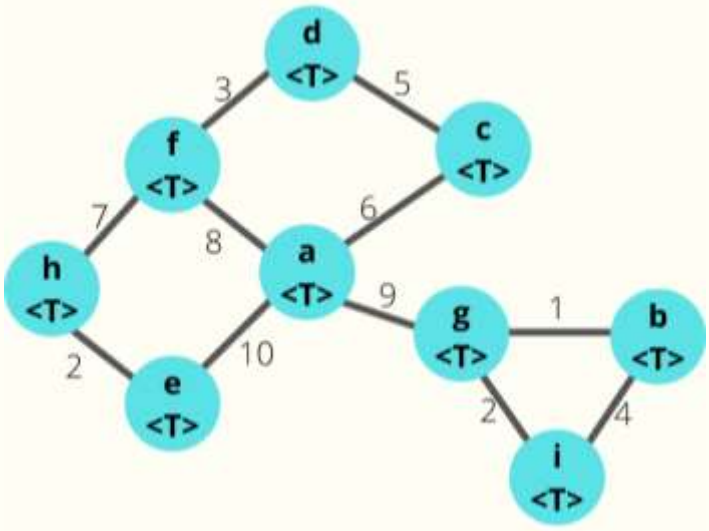
{post: new Graph = {Directed = TRUE, Weighted = TRUE, Vertices = vertices, Edge = vertices-1} }



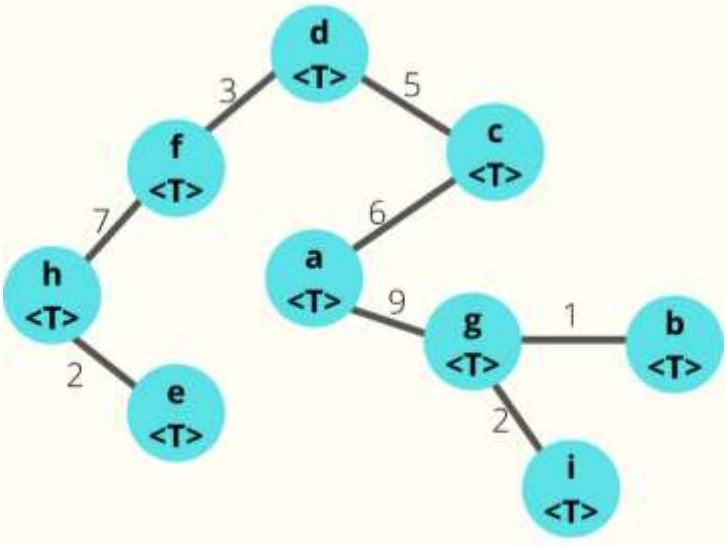
kruskal(graph)

"Returns the minimum spanning tree (MST) of graph"

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