ECE 250 - Project 4
Minimum Spanning Tree using Kruskal's Algorithm
Design Document

Harshit Manchanda, UW UserID: h2mancha

Apr 3rd, 2020

1. Overview

<u>**Class 1:** Node</u>

Description:
It is used to create the sets of different vertices in the Set class which are stored in its member variable list.

Member Variables:
1. Integer vertex
2. Node type pointer next (points to the next node)
3. Node type pointer prev (points to the previous node)

<u>**Class 2:** Set</u>

Description:
It is the Disjoint set class which is created to help find the minimum spanning tree of the graph. It has functions which are used in Kruskal's algorithm to find mst like make set which creates a set for all the vertices, find set which returns the address of a particular set and lastly union which joins to different sets.

Member Variables:
1. A vector name list of type nodeStruct (a struct)

This vector stores the sets of the vertices and joins them when needed.

Member Functions (operations):
1. <u>makeSet:</u> it takes the number of nodes as a parameter and creates sets for each node and stores them in the vector list.
2. <u>unionSet:</u> it joins the two different sets containing the nodes u and v which are passed as parameters.
3. <u>findSet</u>: it returns the address of the head pointer of the vertex passed as parameter.

<u>**Class 3:** MST</u>

Description:
It is a weighted undirected graph represented by an adjacency matrix which stores information about the edges between different vertices and their weights. It has functions like insert edge, delete an edge, finding the degree of a particular vertex, finding the weight of the minimum spanning tree which is achieved using the Set class and edgeWeights vector, etc.

Member Variables:
1. Integer n which holds the number of nodes in the graph
2. Integer numOfEdges which holds the number of edges in the graph
3. An adjacency matrix to keep record of the edges and their weights
4. A vector edgeWeights of type struct graphEdge which is used to hold the sorted edge weights and vertices for that edge

Member Functions (operations):
1. <u>createGraph:</u> resizes the matrix based on the number of nodes sent as parameter and initializes the adjacency matrix with the value 0.0
2. <u>insertEdge:</u> checks if the vertices passed as parameters are in the valid range, if so, then adds that edge between them in the matrix with its weight.
3. <u>deleteEdge</u>: deletes the edge from the matrix if the vertices passed as parameters are valid, otherwise returns failure.
4. <u>degree:</u> returns the number of number of vertices a vertex is connected to by checking in the adjacency matrix.
5. <u>edgeCount:</u> returns the value of the member variable numOfEdges which holds the number of edges
6. <u>clearEdges:</u> traverses through the whole matrix and clears all the edges that exist between the vertices

7. minSpanningTree: it calculates the weight of the minimum spanning tree by using the vector edgeWeights which holds the sorted weights of the edges and then applies Kruskal's algorithm with the help of the class Set, returning the mst value.

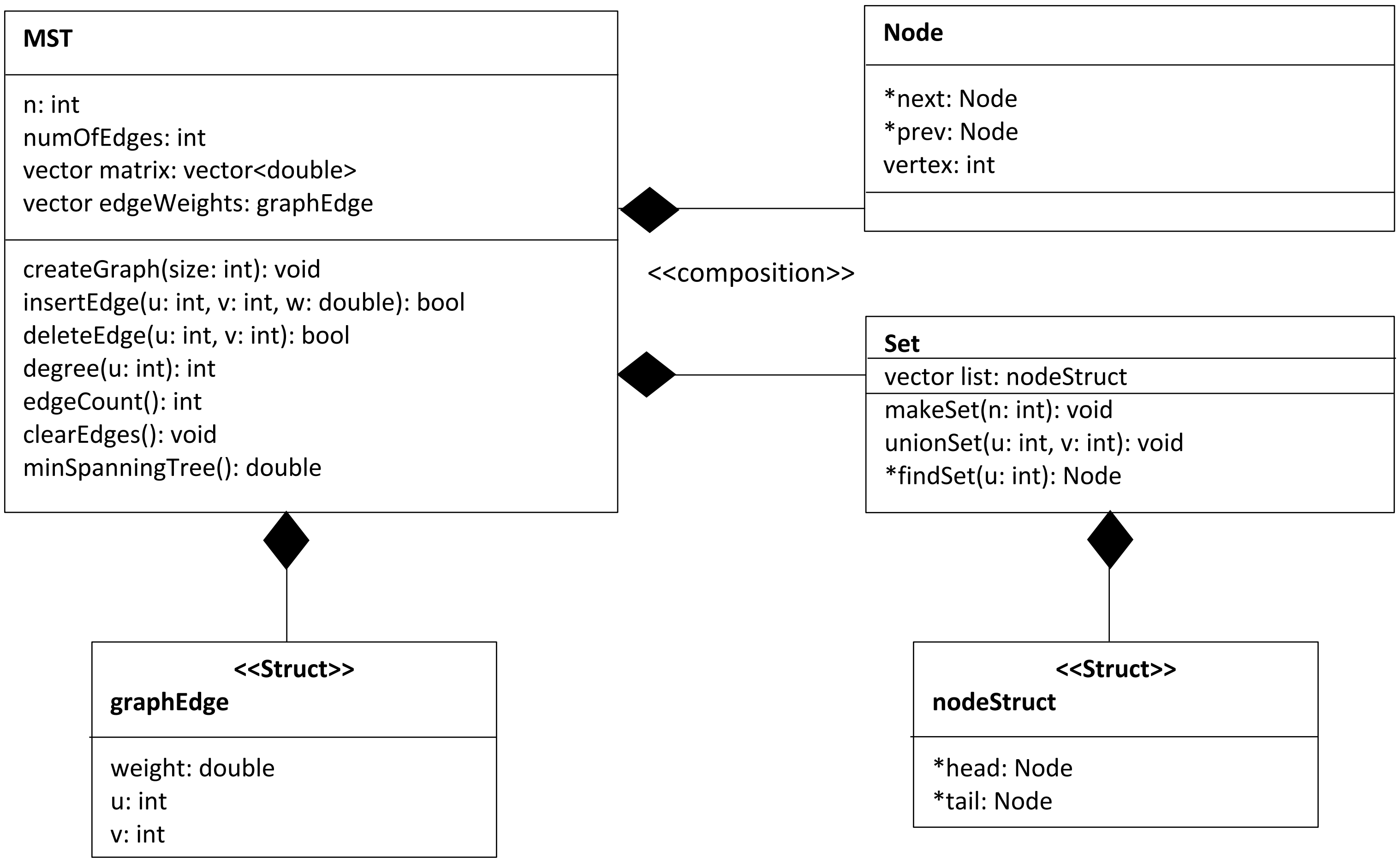
Struct 1: graphEdge
It helps create a vector which will store the edge weights and corresponding vertices to help implement Kruskal's algorithm.

Member Variables:
1. A double weight which stores the weight of the edge
2. Integer u storing one vertex of that edge
3. Integer v storing the other vertex

Struct 2: nodeStruct
This struct also helps create a vector in the Set class which stores the sets of each vertex.

Member Variables:
1. A node type pointer head
2. A node type pointer tail

UML Class Diagram -



2. Constructors/Destructors

Class Node (Constructor):
This parametrised constructor takes in the vertex, address of the next and prev pointers and assigns them to the member variables of that node object

Class Node (Destructor):
The destructor releases the dynamically allocated memory of every node by making the next and prev pointers null.

Class Set (Constructor and Destructor):
This class has the list already as its member so we don't need a constructor and the list is initialised by calling the make set function and we don't need a destructor since there is no dynamically allocated memory.

Class MST (Constructor and Destructor):
The matrix is already a member variable so there is no need to initialise it and in the case of the destructor, since we haven't allocated any memory manually, we don't need to have a destructor.

3. Test Cases

Test 1: Create a new graph, insert edges and check the number of edges. Also check if the degree of a vertex is correct. Now clear the graph and again check for the number of edges.

Test 2: Create another graph, insert edges some correct and some invalid to check if the exceptions are caught or not. Now delete some edges again some valid and some invalid such that the graph is not connected. Lastly, calculate mst.

Test 3: Now, create another graph with all valid edges such that it is connected. Calculate mst to check if the correct value is returned. Now delete some edges and add new ones and calculate mst again.

Test File Example 1:

```
n 4
i 0;1;0.9
i 2;3;1.7
i 3;1;0.5
i 2;0;0.7
edge_count
degree 1
clear
edge_count
```

Test File Example 2:

```
n 4
i 0;1;0.3
i 2;4;1.2
i 3;2;2
i 2;0;-0.7
d 2;3
d 1;2
d 5;0
mst
```

Test File Example 3:

```
n 4
i 0;1;0.9
i 2;3;1.7
i 3;1;0.5
i 2;0;0.7
mst
d 2;0
d 3;2
i 2;1;1.2
mst
```

4. Performance

For class MST:
The create graph function resizes the matrix, so its time complexity is O(V^2) where V is the number of vertices because it's a VxV matrix.
The insert edge function runs in constant time O(1) cause it only consists of conditional statements and setting value in matrix.
The delete edge function also runs in constant time O(1) because it just checks if the edge exists and sets the value 0.0.
The degree function consists of a for loop which runs through the row of that vertex in the matrix to count the edges, so it has a time complexity of O(V).
The edge count function simply returns the value of the member variable so is implemented in constant time O(1).
The clear edges function needs to make all values of the matrix 0.0, so in order to do that we iterate through every element using a nested for loops and thus we get a time complexity of O(V^2).

Lastly, for the mst function, we have used the Set class functions here so we can discuss its complexity with it.
Initially, we first check if the graph is connected or not. In order to do that we iterate through every element of the matrix using nested for loops and break if found disconnected. The worst case run time of it will be O(V^2). After that, we create new Set type object which takes constant time and call the make set function which has a for loop inside it to make a set for every vertex and thus takes linear time O(V). In order to loop on the edges, we first create a vector by iterating through the upper triangular part of the matrix which takes worst case O(V^2) but on average O(E) time because it consists of 2 loops. Then we sort that vector using std::sort which takes an average time of O(ElogE). Finally, we have a loop running through the number of edges and inside it we have a conditional statement calling the find set function which just takes constant time since it returns the head pointer of that vertex, so the number of calls to find set is being made O(E) times. Also, we call the union function of Set class if the condition is satisfied which can take atmost O(logV) time since every time a vertex is moved from one set to another, then size of the new set doubles and thus we get worst case O(logV) and union is being called E times so total time becomes O(ElogV).
We know that the value of E can be at most V^2 and never exceed it, because of that, we can say that O(logE) = O(logV). Moreover, the time complexity of minimum spanning tree also depends on the data structure used which maybe more in the case of the adjacency matrix. Thus, we have the worst time complexity of mst function as O(V^2). However, if we disregard the data structure used for the graph, we get the average time complexity as O(ElogE)/O(ElogV).