CS206A Data Structure

Project3

Student ID: 20160253, 20160338

name: 박예기, 신인철

Part Ⅰ.

We made two classes: Vertex class and Graph class. Vertex class has a constructor and through the constructor, it has a degree as an attribute.

Graph class has seven attributes. We attached under bar(_) in the front of the attributes to express private.

- _count_vertex : The type is integer. It represents the number of vertices.
- _count_edge: The type is integer. It represents the number of vertices.
- _adjacencylist: It is a python list and it consists of lists. The list(index i) consists of tuple(vertex, weight) in which the vertex is adjacent to vertex of index i in _vertex_list.
- _vertex_list: It is a python list and it consists of vertices.
- _edge_list: It is a python list and it consists of edges. Each edge is a tuple:(a vertex, another vertex)
- _vertex_hashtable: It is a python dictionary. The key of this dictionary is a vertex and the value is a list that consists of vertices which are adjacent to the key vertex.
- _edge_hashtable: It is a python dictionary. The key of this dictionary is edge and the value is the weight that corresponding to the key edge.

This Graph class has thirteen public methods.

- WUG(): This function sets _count_vertex and _count_edge to zero, _adjacencylist, _vertex_list, and _edge_list to empty list, and _vertex_hashtable and _edge_hashtable to empty dictionary.
- vertexCount(): It returns _count_vertex.
- edgeCount(): It returns _count_edge.
- getVertices(): It returns a list named "vertices" which consists of the elements of _vertex_list.
- addVertex(Object): This function adds one to _count_vertex and appends the object to _vertex_list. And, it appends an empty list to _adjacencylist and add (key, value) = (object,

empty list) to _vertex_hashtable.

- removeVertex(Object): It subtracts degree of vertex to _count_edge and subtracts one to _count_vertex. It has for-loop in neighbors of the object. In the for-loop, it removes object in _vertex_hashtable which has the object in the value and it deletes edges which contains the object in _edge_hashtable and _edge_list. It removes (object, weigth) in _adjacencylist. It removes adjacency list of object in _adjacencylist and removes object in _vertex_list and deletes vertex key in in _vertex_hashtable.
- isVertex(Object): It returns whether the object is a key of _vertex_hashtable or not.
- degree(Object): It returns the degree of the object.
- getNeighbors(Object): It returns a list named "neighbors" which consists of the elements of the list which is the value of key object of _vertex_hashtable
- addEdge(Object, Object, int): This function adds one to degree of each object and _count_edge. It appends (object, int) (=(vertex, weight)) to _adjacencylist of each object. It appends (object, object) to _edge_list. And it adds each vertex to the list of the other key vertex of _vertex_hashtable and adds (key, value) = ((object, object), weight) to _edge_hashtable.
- removeEdge(Object, Object): This function subtracts one to degree of each object and _count_edge. It removes (object, object) in _edge_list and delete the key (object, object) in _edge_hashtable. And then, it removes (object, weight) in each _adjacencylist.
- isEdge(Object, Object): This function returns whether (object, object) is a key of _edge_hashtable or not.
- weight(Object, Object): This function returns the value of the key (object, object) in _edge_hashtable.

The below picture is test case.

```python
v1 = Vertex()
v2 = Vertex()
v3 = Vertex()
v4 = Vertex()
v5 = Vertex()
v6 = Vertex()

g = Graph()
g.WUG()
g.addVertex(v1)
g.addVertex(v2)
g.addVertex(v3)
g.addVertex(v4)
g.addVertex(v5)
g.addVertex(v6)
g.addEdge(v1, v2, 5)
g.addEdge(v2, v3, 4)
g.addEdge(v2, v5, 2)
g.addEdge(v6, v3, 3)
g.addEdge(v5, v4, 4)
```

```python
print("The vertices of the graph g is")
print(g.getVertices(), "₩n")
print("The neighbors of vertex v2 is")
print(g.getNeighbors(v2), "₩n")

g.removeEdge(v5, v2)
g.removeVertex(v3)

print("The degree of vertex v2 is")
print(v2.degree, "₩n")
print("Is edge(v1, v2) in the graph?")
print(g.isEdge(v1, v2), "₩n")
print("Is edge(v2, v5) in the graph?")
print(g.isEdge(v2, v5), "₩n")
print("The weight of edge(v4, v5) is")
print(g.weight(v4, v5))
```

```
The vertices of the graph g is
[<__main__.Vertex object at 0x03672170>, <__main__.Vertex object at 0x036728B0>, <__main__.Vertex object at 0x036728F0>, <__main__.Vertex object at 0x03672910>, <__main__.Vertex object at 0x03672970>, <__main__.Vertex

The neighbors of vertex v2 is
[<__main__.Vertex object at 0x03672170>, <__main__.Vertex object at 0x036728F0>, <__main__.Vertex object at 0x03672970>]

The degree of vertex v2 is
2

Is edge(v1, v2) in the graph?
True

Is edge(v2, v5) in the graph?
False

The weight of edge(v4, v5) is
4
```

Part Ⅱ.

I add function minSpanTree() in class Graph(). We use kruskal's algorithm to make graph as a tree. Also we have to check that total number of edges is V-1. If not, we can't make tree. Time complexity of kruskal's algorithm is O(|e| log|e|). Also time complexity of calculating number of edges is O(|V|). So, total time complexity is O(|e| log|e| +|V|). The below pictures are two test cases.

```
v =[0 for col in range(7)
v[1] = Vertex()
v[2] = Vertex()
v[3] = Vertex()
v[4] = Vertex()
v[5] = Vertex()
v[6] = Vertex()

g = Graph()
g.WUG()
g.addVertex(v[1])
g.addVertex(v[2])
g.addVertex(v[3])
g.addVertex(v[4])
g.addVertex(v[5])
g.addVertex(v[6])
g.addEdge(v[1], v[2], 5)
g.addEdge(v[2], v[3], 4)
g.addEdge(v[2], v[5], 2)
g.addEdge(v[6], v[3], 3)
g.addEdge(v[5], v[4], 4)
g.addEdge(v[1], v[3], 1)
g.addEdge(v[1], v[6], 1)
```

Case1. When there is spanning tree.

Run wug

```
C:\Anaconda3\python.exe "C:/Users/In Chul Shin/PycharmProjects/untitled7/wug.py"
I change graph as minimun spanning tree using Kruskal's Algorithm

vertex V[1] and V[3] are connected and value of edge is 1

vertex V[1] and V[6] are connected and value of edge is 1

vertex V[5] and V[2] are connected and value of edge is 2

vertex V[3] and V[2] are connected and value of edge is 4

vertex V[5] and V[4] are connected and value of edge is 4


Process finished with exit code 0
```

```
v =[0 for col in range(7)]
v[1] = Vertex()
v[2] = Vertex()
v[3] = Vertex()
v[4] = Vertex()
v[5] = Vertex()
v[6] = Vertex()

g = Graph()
g.WUG()
g.addVertex(v[1])
g.addVertex(v[2])
g.addVertex(v[3])
g.addVertex(v[4])
g.addVertex(v[5])
g.addVertex(v[6])
g.addEdge(v[1], v[2], 5)
g.addEdge(v[2], v[3], 4)
g.addEdge(v[1], v[3], 2)
g.addEdge(v[6], v[5], 3)
g.addEdge(v[5], v[4], 4)
g.addEdge(v[4], v[6], 1)
```

Case2. When there is no spanning tree.