GTU Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework #8 Report

Sena Nur Ulukaya
1901042622

1. SYSTEM REQUIREMENTS

a. Non-Functional Requirements

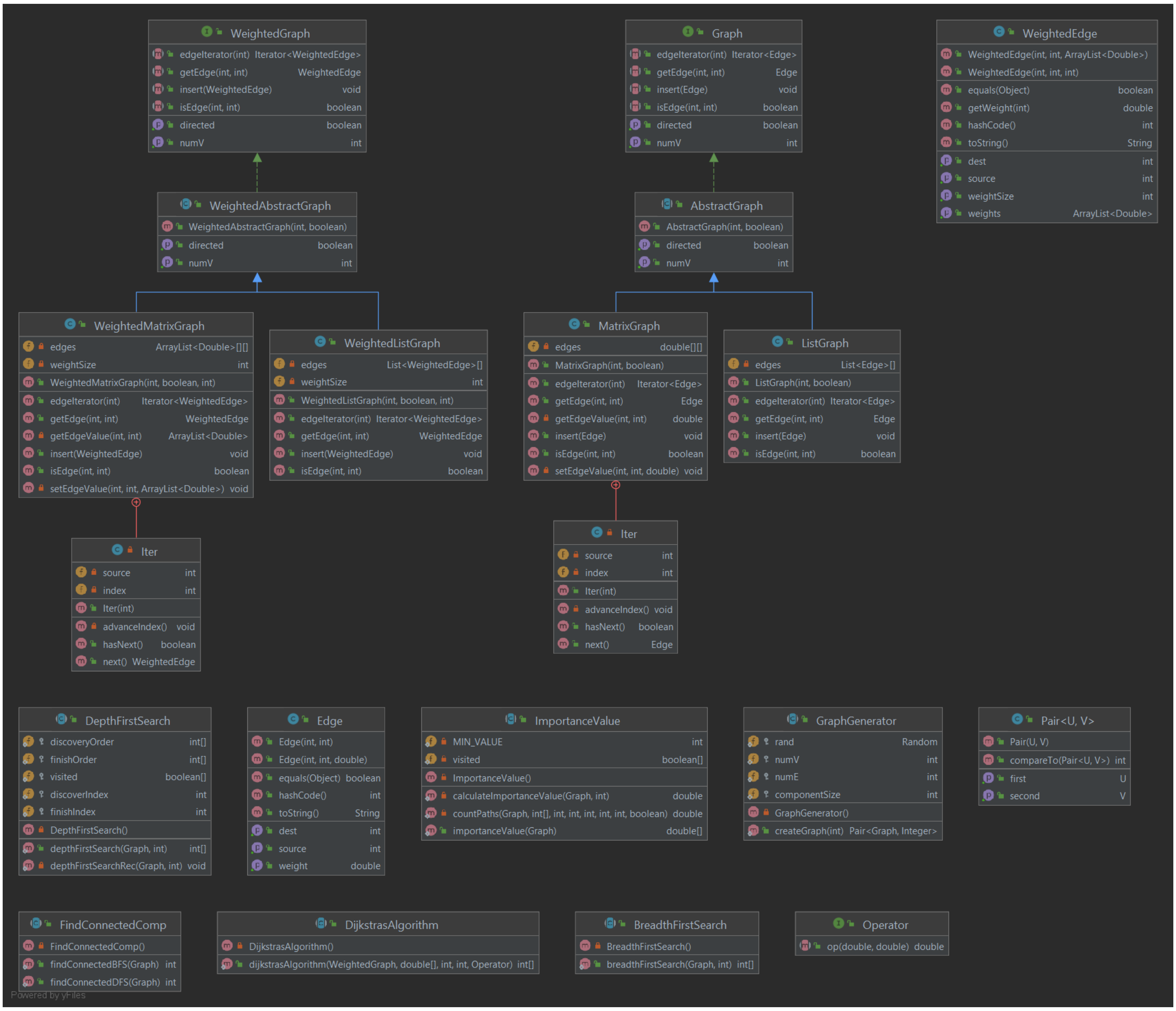
- Java Runtime Environment to run code.
- Some memory to test the program because it will test with different data sizes. (100-200 KB)
- Min 4 GB RAM because it will load some data and can be difficult with lower RAM.
- Java as programming language.

b. Functional Requirements

- Method to generalizate of the implementation of Dijkstra's algorithm in the book with the several weight properties and different associative operators.
  - WeightedGraph class to represent graphs with a several weight properties.
  - Operator class to represent associative operators.
  - It should run efficiently for both ListGraph and MatrixGraph representations.

- Method to find connected component numbers in a graph with the BFS algorithm.
- Method to find connected component numbers in a graph with the DFS algorithm.
- Method to generate a graph with different sizes and various sparsity.
- Method to calculate run times.
- Method to calculate importance value of the all vertex of a graph.
  - Method to count shortest paths and shortest paths with the given vertex V.
  - Method to calculate importance value of a vertex.

2. CLASS DIAGRAMS



3. PROBLEM SOLUTION APPROACH

- **Part1: Generalization of the implementation of Dijkstra's algorithm**
  - I changed the Graph implementation of the book and create a new WeightedGraph. The reason for that I need to have a generic weight array. I used arrayList for my weight array. WeightedGraph is constructed with a weight array size which is additional info to Graph. Also, WeightedEdge is constructed with an ArrayList to get the values for the weights. According to that I have updated the necessary parts of the Graph implementation to fit my weight array.
  - You can hold as much properties for the weight.
  - I have used a priority queue to make my method run efficiently for both ListGraph and MatrixGraph representations of the graph. In that way, it doesn't have to search for the minimum distance value, because priority queue already keeps the minimum value in its head value.
  - I have used a Pair class in the Priority queue to keep vertex itself and its distance, to get the min value easily. I also used a comparator to compare distances in the priority queue constructor which looks for pair's second values.
  - I have used a Operator class for generic associative operators. In that way we can give any associative operator as a parameter to the Dijkstra's algorithm. It has a abstract op() method in it which I implemented while testing.

- **Part2: Find the number of connected components in a graph**
  - I used the implementation of DFS and BFS from the book.
  - I called the algorithms until there's no -1 in the returned array. If it's -1 it is not a part of the component. So, I count them as a different component and marked the others as true to not to check them again.
  - If the size of the graph is 0 or 1, connected component is the size.
  - To test I implemented a class to generate a graph with the given size. This class also count the number of components while generating so I can test if my methods run properly. I implemented this class with different number of edges to generate various sparsity.

- **Part3: The importance of a vertex v**
  - Firstly, I have counted the shortest paths and shortest paths with the v recursively between the given start and end vertexes. If I can find shortest path in the search I updated. I returned the number of shortest path with given vertex v/number of shortest paths in this method if it is defined (no 0/0 or x/0). Otherwise I throwed an exception and count its value as 0.
  - After I found the number of shortest path with given vertex v/number of shortest paths I summed them for all the connected components's start and end values. I

found the connected component array with the DFS. Then I divided the sum into the connected component count.

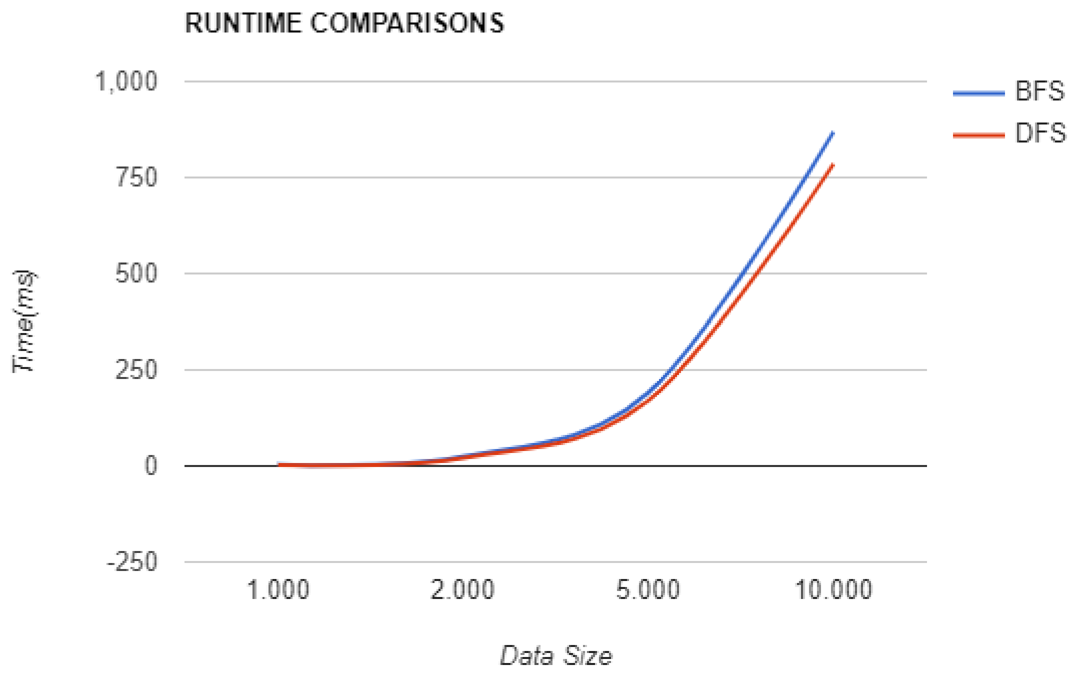
- Lastly, I calculated for all the vertices of the graph and returned a double array.
- If graph size is less than 3, formula can't be used so I throwed an exception. Also, I counted value 0 if connected component count is less than 3.

4. TEST CASES

- **Part1:**
  - I created 2 graphs one with matrix, other is list representation. I used the weight array size as 3, and added different 3 values for each edge both graphs.
  - Then I iterate through edges to show.
  - After that I displayed the results for each case.
    With all the weight array indexes for 4 different operators. I used the +,x,*,- operators but we can define any operation with the Operator class.
    For – case, an exception has throwned because it is not associative.
    I also give null values as parameter and catched an exception.

- **Part2:**
  - Firstly, I tested the accuracy of my methods with DFS, BFS and GraphGeneator methods to show they all give the same result.
  - Then, I displayed the results of the run time comparisons for the sizes 1000,2000,5000,1000 in ms for both DFS and BFS.
  - Results are in the table:

Sizes/Running Time(ms)	**BFS**	**DFS**
1000	3.3	2.2
2000	24.3	20.8
5000	190.8	170.2
10000	868.4	785



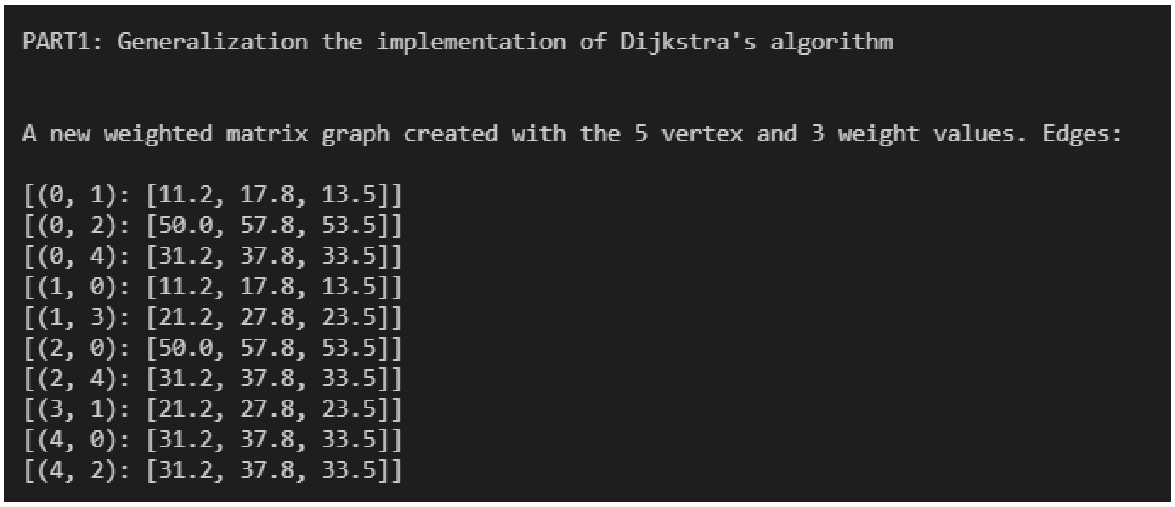
- The graphs and table show that DFS is slightly better than DFS in term of the running times.
- Each of them takes more time when size is greater.

- **Part3:**
  - Firstly I have displayed the result of the each vertices importance value for the graph with size of 8 and 5.
  - Then I tested my method with the size of 200 which generated by my GraphGenerator. I run it 3 times and showed it has successfully passed the test.
  - Also, as I mentioned I throwed an exception when graph size is smaller than 3, because formula won't work.

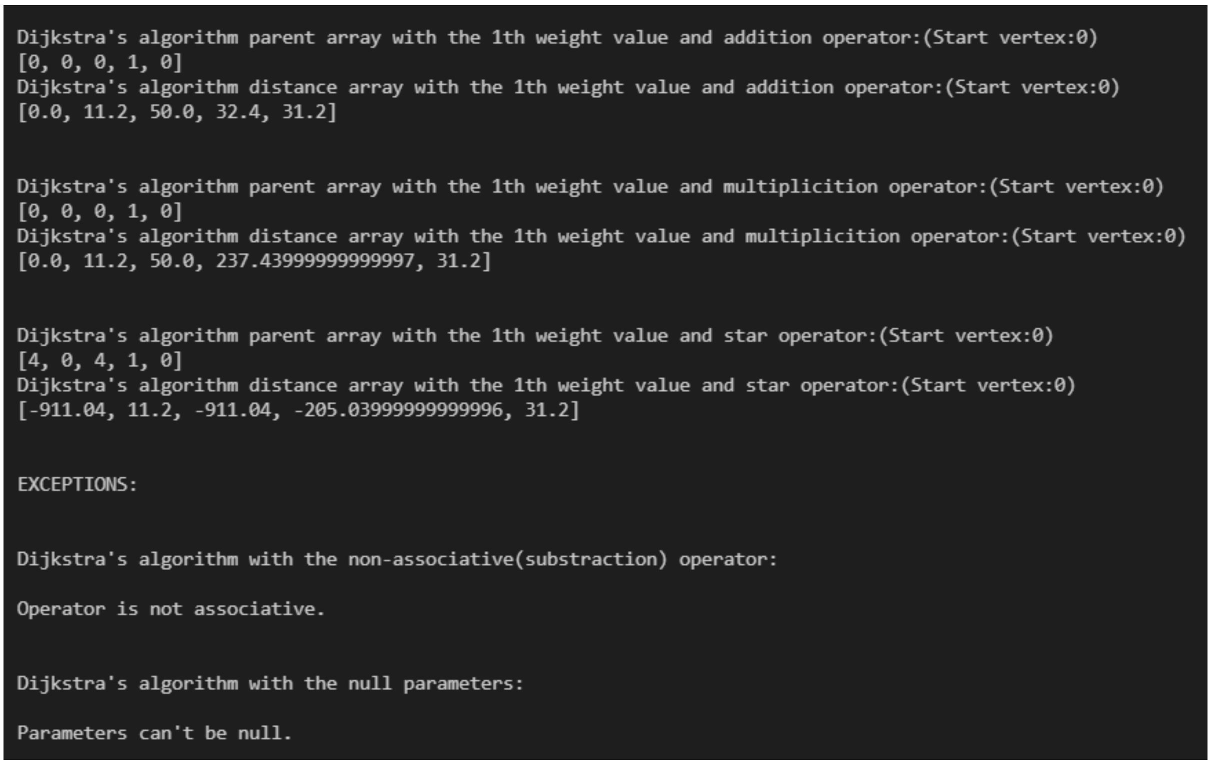
5. RUNNING AND RESULTS

PART1

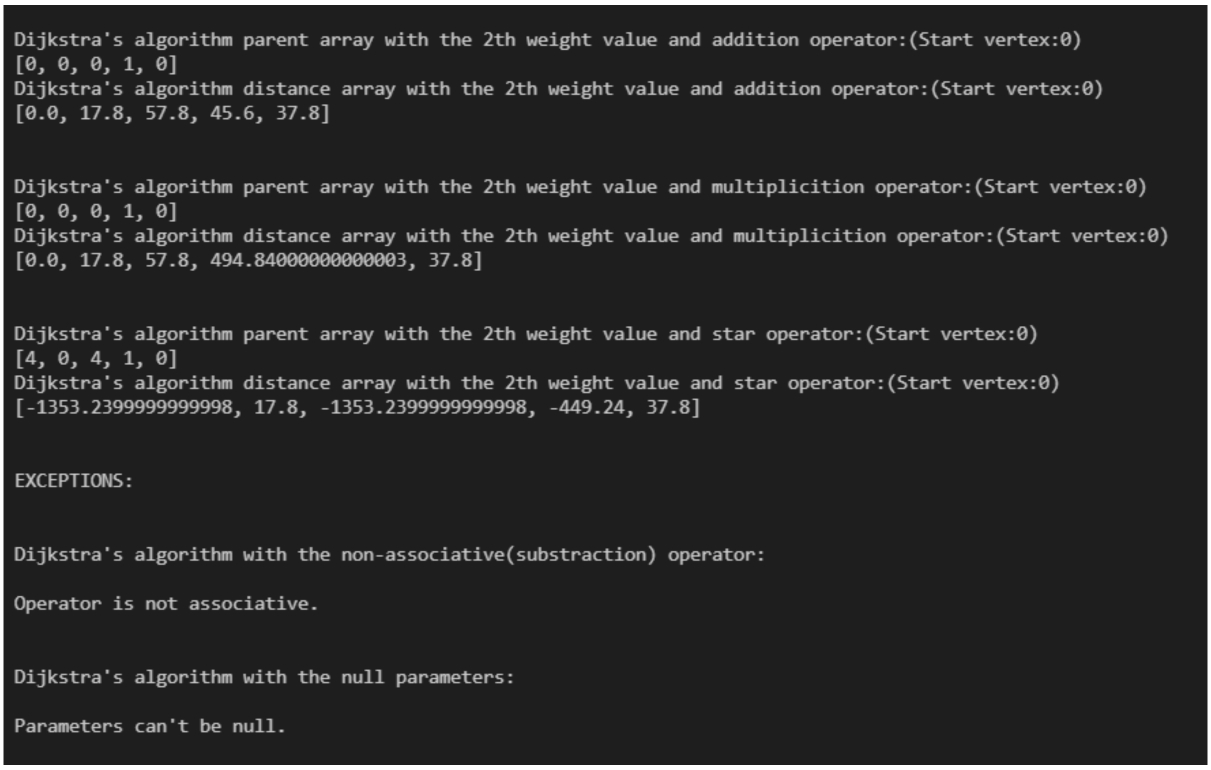
- **Weighted MatrixGraph**



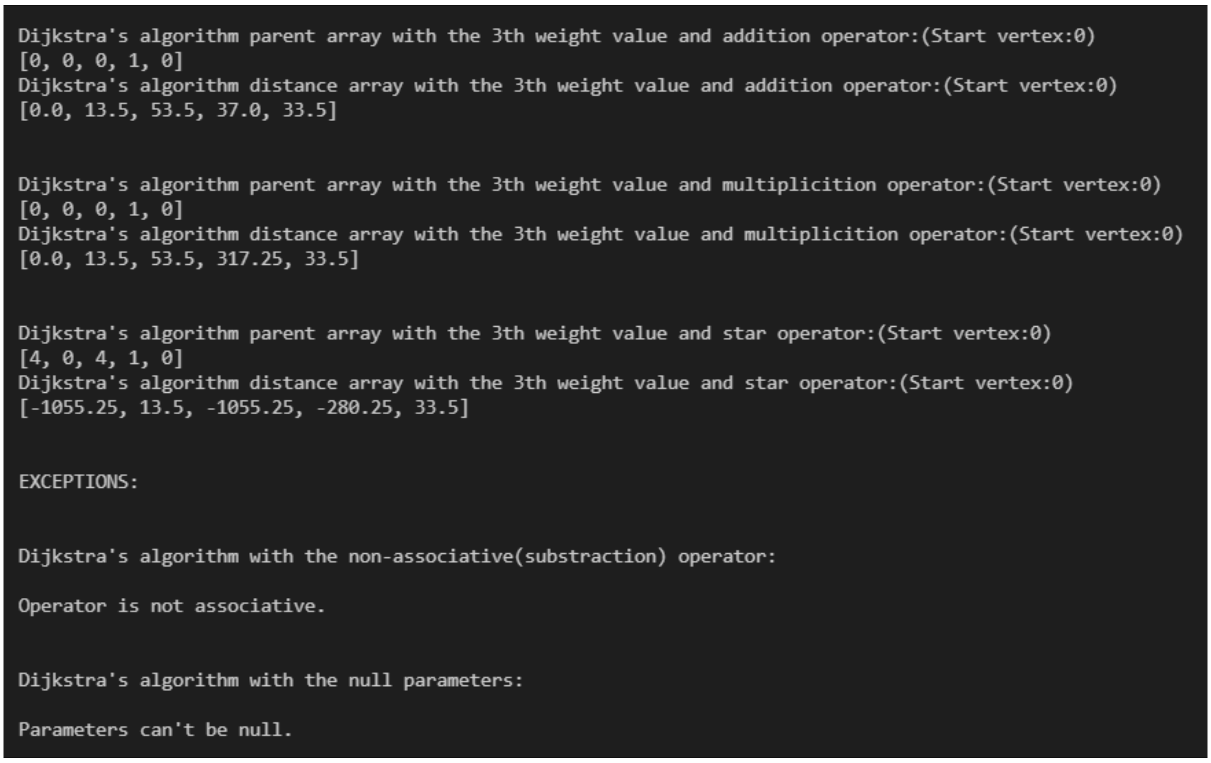
```
PART1: Generalization the implementation of Dijkstra's algorithm


A new weighted matrix graph created with the 5 vertex and 3 weight values. Edges:

[(0, 1): [11.2, 17.8, 13.5]]
[(0, 2): [50.0, 57.8, 53.5]]
[(0, 4): [31.2, 37.8, 33.5]]
[(1, 0): [11.2, 17.8, 13.5]]
[(1, 3): [21.2, 27.8, 23.5]]
[(2, 0): [50.0, 57.8, 53.5]]
[(2, 4): [31.2, 37.8, 33.5]]
[(3, 1): [21.2, 27.8, 23.5]]
[(4, 0): [31.2, 37.8, 33.5]]
[(4, 2): [31.2, 37.8, 33.5]]
```



```
Dijkstra's algorithm parent array with the 1th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and addition operator:(Start vertex:0)
[0.0, 11.2, 50.0, 32.4, 31.2]


Dijkstra's algorithm parent array with the 1th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and multiplicition operator:(Start vertex:0)
[0.0, 11.2, 50.0, 237.43999999999997, 31.2]


Dijkstra's algorithm parent array with the 1th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and star operator:(Start vertex:0)
[-911.04, 11.2, -911.04, -205.03999999999996, 31.2]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

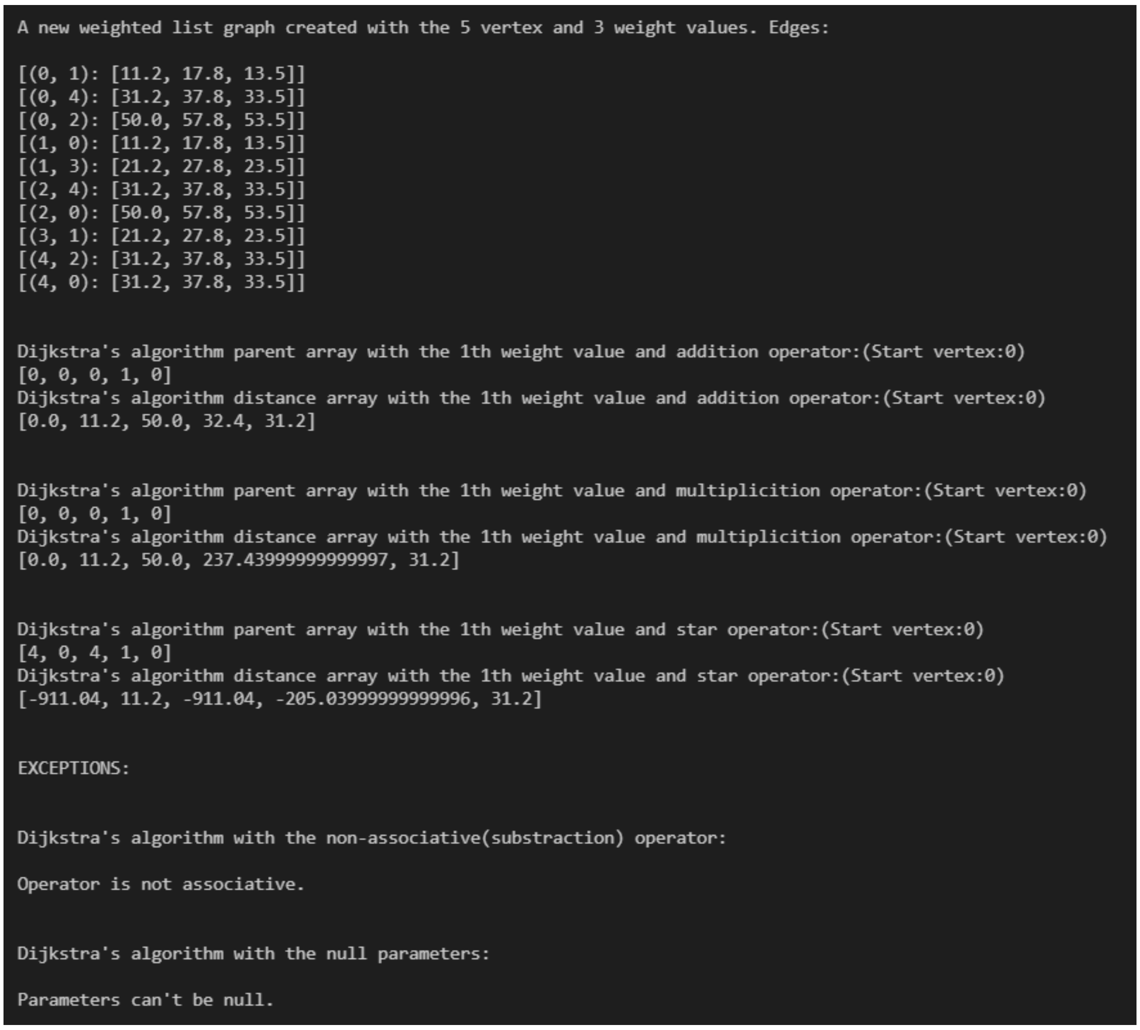


```
Dijkstra's algorithm parent array with the 2th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and addition operator:(Start vertex:0)
[0.0, 17.8, 57.8, 45.6, 37.8]


Dijkstra's algorithm parent array with the 2th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and multiplicition operator:(Start vertex:0)
[0.0, 17.8, 57.8, 494.84000000000003, 37.8]


Dijkstra's algorithm parent array with the 2th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and star operator:(Start vertex:0)
[-1353.2399999999998, 17.8, -1353.2399999999998, -449.24, 37.8]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

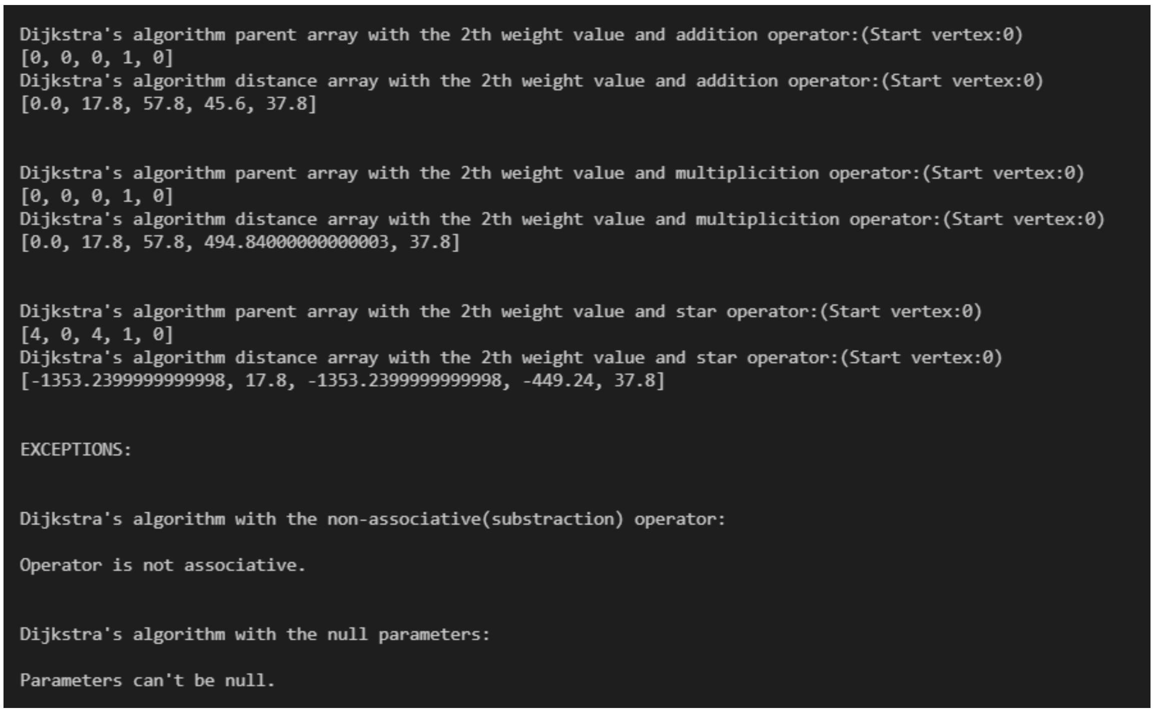


```
Dijkstra's algorithm parent array with the 3th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and addition operator:(Start vertex:0)
[0.0, 13.5, 53.5, 37.0, 33.5]


Dijkstra's algorithm parent array with the 3th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and multiplicition operator:(Start vertex:0)
[0.0, 13.5, 53.5, 317.25, 33.5]


Dijkstra's algorithm parent array with the 3th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and star operator:(Start vertex:0)
[-1055.25, 13.5, -1055.25, -280.25, 33.5]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

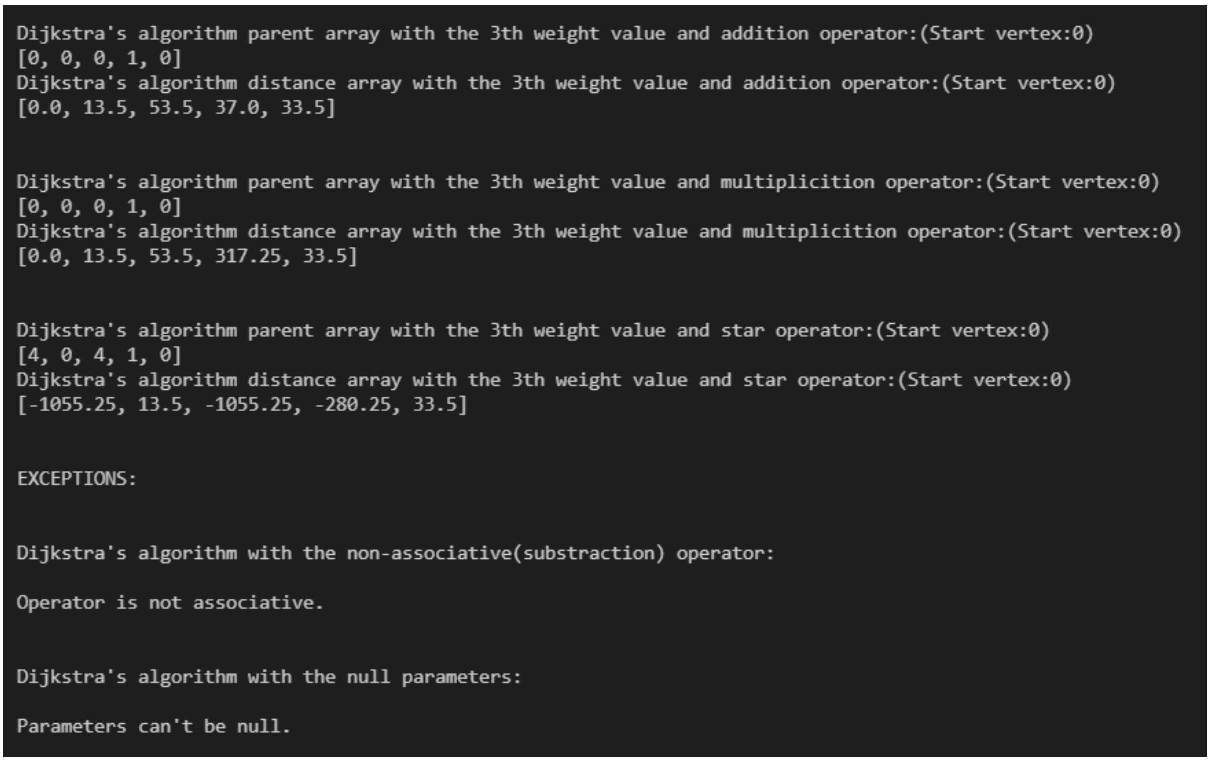
- **Weighted ListGraph**



```
A new weighted list graph created with the 5 vertex and 3 weight values. Edges:

[(0, 1): [11.2, 17.8, 13.5]]
[(0, 4): [31.2, 37.8, 33.5]]
[(0, 2): [50.0, 57.8, 53.5]]
[(1, 0): [11.2, 17.8, 13.5]]
[(1, 3): [21.2, 27.8, 23.5]]
[(2, 4): [31.2, 37.8, 33.5]]
[(2, 0): [50.0, 57.8, 53.5]]
[(3, 1): [21.2, 27.8, 23.5]]
[(4, 2): [31.2, 37.8, 33.5]]
[(4, 0): [31.2, 37.8, 33.5]]


Dijkstra's algorithm parent array with the 1th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and addition operator:(Start vertex:0)
[0.0, 11.2, 50.0, 32.4, 31.2]


Dijkstra's algorithm parent array with the 1th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and multiplicition operator:(Start vertex:0)
[0.0, 11.2, 50.0, 237.43999999999997, 31.2]


Dijkstra's algorithm parent array with the 1th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 1th weight value and star operator:(Start vertex:0)
[-911.04, 11.2, -911.04, -205.03999999999996, 31.2]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

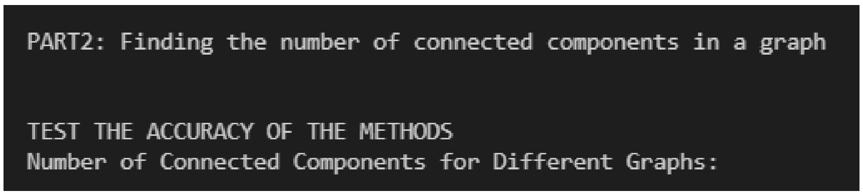


```
Dijkstra's algorithm parent array with the 2th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and addition operator:(Start vertex:0)
[0.0, 17.8, 57.8, 45.6, 37.8]


Dijkstra's algorithm parent array with the 2th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and multiplicition operator:(Start vertex:0)
[0.0, 17.8, 57.8, 494.84000000000003, 37.8]


Dijkstra's algorithm parent array with the 2th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 2th weight value and star operator:(Start vertex:0)
[-1353.2399999999998, 17.8, -1353.2399999999998, -449.24, 37.8]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

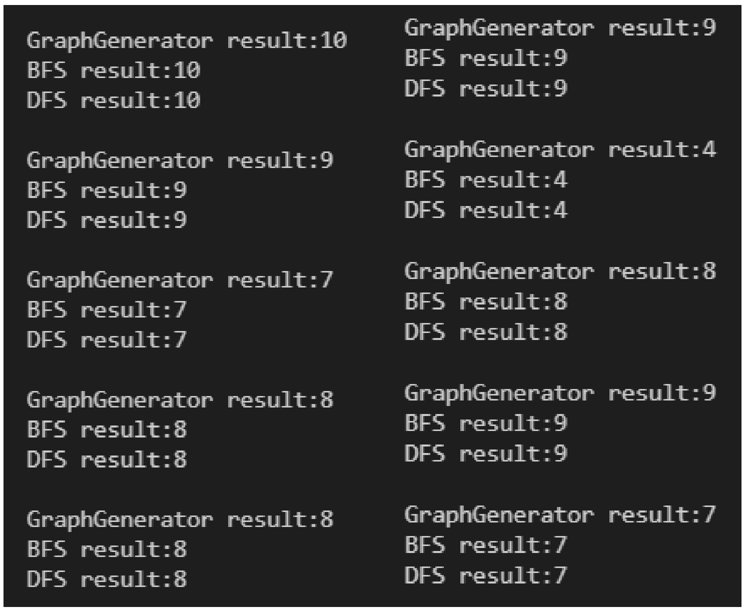


```
Dijkstra's algorithm parent array with the 3th weight value and addition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and addition operator:(Start vertex:0)
[0.0, 13.5, 53.5, 37.0, 33.5]


Dijkstra's algorithm parent array with the 3th weight value and multiplicition operator:(Start vertex:0)
[0, 0, 0, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and multiplicition operator:(Start vertex:0)
[0.0, 13.5, 53.5, 317.25, 33.5]


Dijkstra's algorithm parent array with the 3th weight value and star operator:(Start vertex:0)
[4, 0, 4, 1, 0]
Dijkstra's algorithm distance array with the 3th weight value and star operator:(Start vertex:0)
[-1055.25, 13.5, -1055.25, -280.25, 33.5]


EXCEPTIONS:


Dijkstra's algorithm with the non-associative(substraction) operator:

Operator is not associative.


Dijkstra's algorithm with the null parameters:

Parameters can't be null.
```

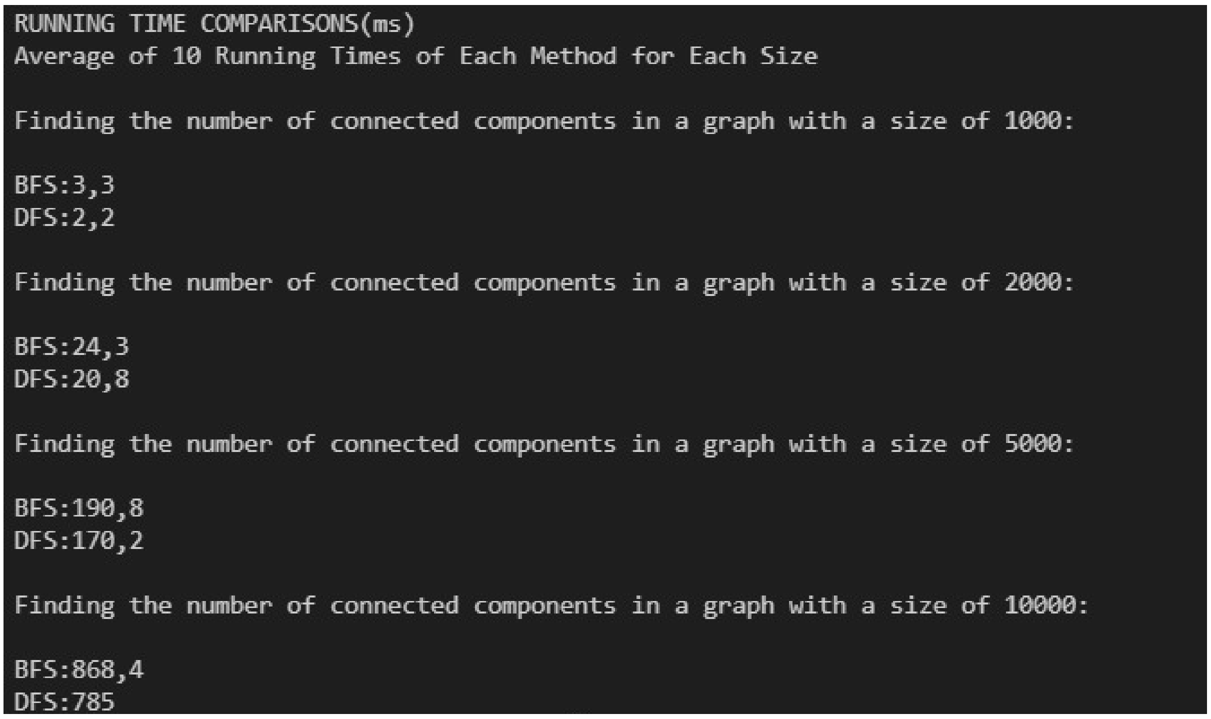
PART2



```
PART2: Finding the number of connected components in a graph


TEST THE ACCURACY OF THE METHODS
Number of Connected Components for Different Graphs:
```

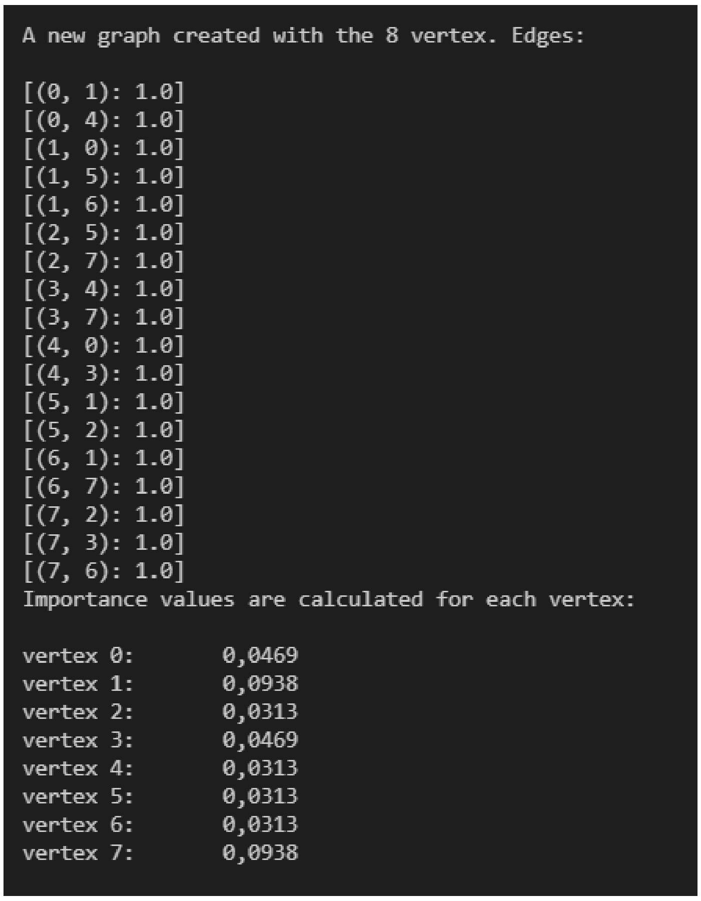


```
GraphGenerator result:10    GraphGenerator result:9
BFS result:10               BFS result:9
DFS result:10               DFS result:9

GraphGenerator result:9     GraphGenerator result:4
BFS result:9                BFS result:4
DFS result:9                DFS result:4

GraphGenerator result:7     GraphGenerator result:8
BFS result:7                BFS result:8
DFS result:7                DFS result:8

GraphGenerator result:8     GraphGenerator result:9
BFS result:8                BFS result:9
DFS result:8                DFS result:9

GraphGenerator result:8     GraphGenerator result:7
BFS result:8                BFS result:7
DFS result:8                DFS result:7
```

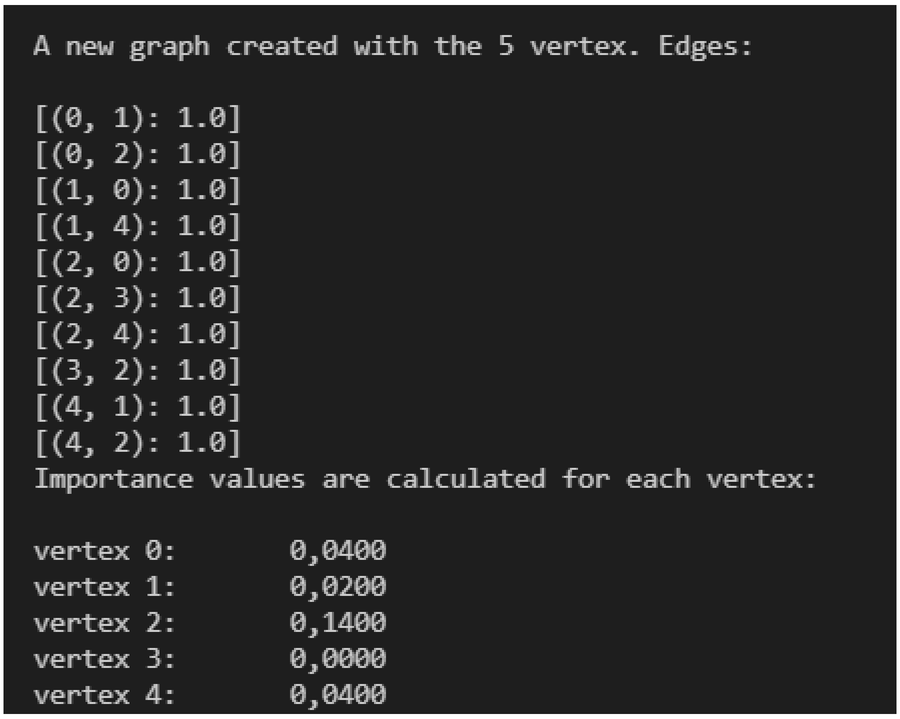


```
RUNNING TIME COMPARISONS(ms)
Average of 10 Running Times of Each Method for Each Size

Finding the number of connected components in a graph with a size of 1000:

BFS:3,3
DFS:2,2

Finding the number of connected components in a graph with a size of 2000:

BFS:24,3
DFS:20,8

Finding the number of connected components in a graph with a size of 5000:

BFS:190,8
DFS:170,2

Finding the number of connected components in a graph with a size of 10000:

BFS:868,4
DFS:785
```

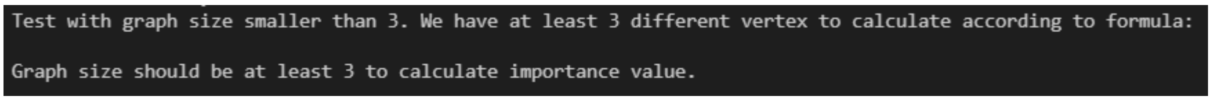
PART3



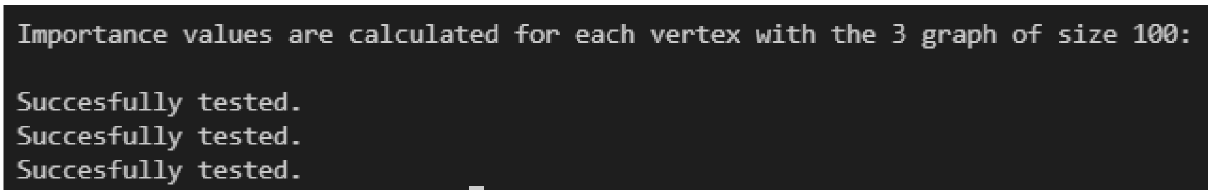
```
A new graph created with the 8 vertex. Edges:

[(0, 1): 1.0]
[(0, 4): 1.0]
[(1, 0): 1.0]
[(1, 5): 1.0]
[(1, 6): 1.0]
[(2, 5): 1.0]
[(2, 7): 1.0]
[(3, 4): 1.0]
[(3, 7): 1.0]
[(4, 0): 1.0]
[(4, 3): 1.0]
[(5, 1): 1.0]
[(5, 2): 1.0]
[(6, 1): 1.0]
[(6, 7): 1.0]
[(7, 2): 1.0]
[(7, 3): 1.0]
[(7, 6): 1.0]
Importance values are calculated for each vertex:

vertex 0:       0,0469
vertex 1:       0,0938
vertex 2:       0,0313
vertex 3:       0,0469
vertex 4:       0,0313
vertex 5:       0,0313
vertex 6:       0,0313
vertex 7:       0,0938
```



```
A new graph created with the 5 vertex. Edges:

[(0, 1): 1.0]
[(0, 2): 1.0]
[(1, 0): 1.0]
[(1, 4): 1.0]
[(2, 0): 1.0]
[(2, 3): 1.0]
[(2, 4): 1.0]
[(3, 2): 1.0]
[(4, 1): 1.0]
[(4, 2): 1.0]
Importance values are calculated for each vertex:

vertex 0:       0,0400
vertex 1:       0,0200
vertex 2:       0,1400
vertex 3:       0,0000
vertex 4:       0,0400
```



```
Test with graph size smaller than 3. We have at least 3 different vertex to calculate according to formula:

Graph size should be at least 3 to calculate importance value.
```



```
Importance values are calculated for each vertex with the 3 graph of size 100:

Succesfully tested.
Succesfully tested.
Succesfully tested.
```