BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter
(Group Work)

Student Information:	*Surname*	*Given Names*	*Student ID Number*
	1. Nofarditya	Arkaan	2101718425
	2. Bismadhika	Fiqhy	2101714824
	3. Kamal	Fauzan	2101720700

Course Code	:COMP6340	Course Name	: Analysis of Algorithm
Class	:L3BC	Name of Lecturer(s)	: 1. Ms. Maria Seraphina
Major	:Computer Science		

Title of Assignment :
(if any)

Type of Assignment :Final Report Paper

Submission Pattern

Due Date	:15th January 2019	Submission Date	:14th January 2019

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating
BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality
By signing this assignment, we understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith we declare that the work contained in this assignment is our own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

1. Introduction

- Background

  Mail courier is an employee of a postal service who delivers mail and package to both residences and businesses. Most of the carrier in Indonesia use motorcycle to reach address with long distances while reducing delivery time. It would be more efficient in time and resource if every mailman creates the shortest route to deliver all the mails beforehand.

- Problems

  A mail courier may have multiple delivery destinations and there are routes a courier can take to visit all the destinations. Not all route options are the most efficient.

- Aims

  The most efficient route for the mail courier to both deliver all packages to each address and return to the post office.

- Benefits

  To allow the postal service delivers more mails in less time and fuel therefore granting more efficiency in their service.

- Features
  - Create available number of addresses.
  - Pick which address to have a connected path with each other.
  - Create distance of each path.
  - Graph of the minimum spanning tree for the mailman to reach all addresses with kruskal's algorithm.
  - Graph of the shortest path for the mailman to return to the post office with Dijkstra's algorithm.

2. Related Work

Google maps, Waze, and any other program that implements pathfinding algorithms.

3. Implementation

- Formal description of the problem

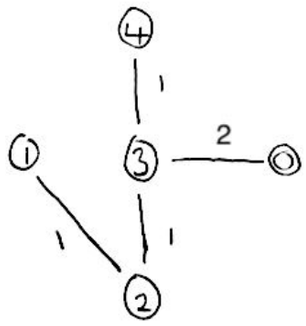
  The starting point of the mail courier will always be the post office, but the route may have multiple destinations. Because of this, the mail courier must find the most efficient way to deliver mail in a given route.

- Design of algorithm

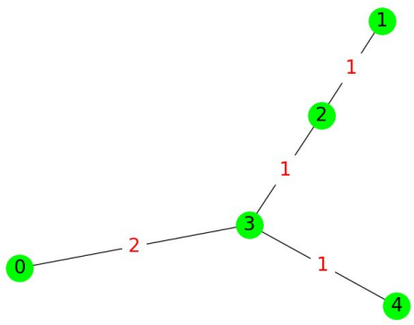
  In this project, we implemented 3 algorithms:
  - Kruskal's algorithm
    - Used to produce a Minimum Spanning Tree (MST) of the original graph
  - Dijkstra's algorithm
    - Used to find the shortest path between two nodes.
    - To find the shortest route for the mail courier to return to the post office
  - Depth First Search (DFS)
    - Visits all nodes and logs the path to visit those nodes
    - To give the path or route for the mail courier to visit all the destinations

- Proof of correctness
  Compared the graphs generated by the program with the graphs calculated manually using a few preset graphs.

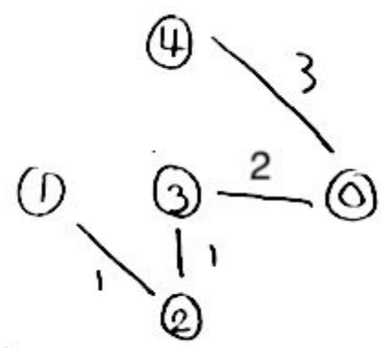
  Graph 1
  Manual (MST)



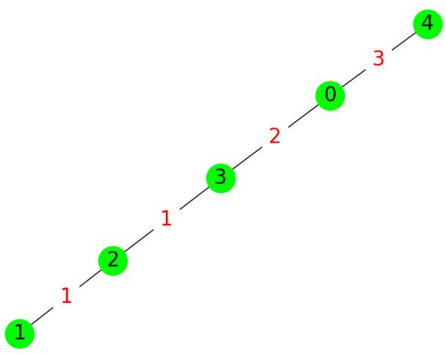
  Program (MST)



  Manual (SPT)



  Program (SPT)



  We can see that both graphs match one another meaning that the program is outputting the correct graphs.

  We also checked the path generated by the program and compared it with the path we calculated manually and the program outputs the same results as well, meaning the program outputs the correct path as well.

- Complexity Analysis
  Time Complexity in microseconds

N	DFS	Kruskal(MST)	Dijkstra(SPT)
5	109	103	139
50	5554	5551	15538
100	21104	22105	63552
1000	2019068	2078689	6888932

  Space Complexity in bytes

N	DFS	Kruskal	Dijkstra
5	328	318	166
50	1404	4418	554
100	2396	14866	1002
1000	21330	1520114	8838

4. Evaluation

- Theoretical analysis of the algorithms
  - Depth First Search

    DFS is a recursive algorithm that uses stack movement and backtracking to avoid infinite looping. The algorithm will do an exhaustive searches of all nodes by going ahead, if possible by backtracking. In backtracking, if there are no possible nodes left the program will move backwards to find nodes to traverse.
  - Kruskal's Minimum Spanning Tree

    Kruskal's algorithm is a minimum spanning tree algorithm which finds the minimum edge value that connects each two different vertices and forms a tree that include every vertices.
  - Dijkstra's Shortest Path Tree

    Dijkstra's algorithm is a shortest path tree algorithm, which is a tree that will always give the shortest path from one node to any node.

- Implementation Details
  1. DFS greedy pathfinding

```
def pathFind(graph_input, dijkstra_true=False, start=0, end=0):
    graph = []
    for i in range(len(graph_input)):
        graph.append(graph_input[i][:])
```

define function pathFind, append all elements in the selected or generated input graph to different graph list within the function.

```
    pathfind = True
    vertices = list(range(len(graph)))
    path = [start]
    visited = [start]
    backtrack = [start]
```

Pathfind boolean to stop while loop,
Path to store the visit alkl nodes path,
Visited to store already visited nodes,
Backtrack to keep track movement and allowing backtracking while pathfinding.

```
while pathfind:
    currentver = backtrack[-1]
    nextver = graph[currentver].index(min(graph[currentver]))

    if dijkstra_true and nextver == end and min(graph[currentver]) != inf:
        path.append(nextver)
        break

    if ((set(vertices)&set(visited))==set(vertices)):
        pathfind=False
```

Loop pathfinding, stops if all elements in vertices list is the same with elements in visited list.
Currentver is used to identify current node in path, while nextver value is the next node decided with the lowest value of edge to any possible unvisited node.
If the nextver is not inf then append the next node to path list, if yes then pop last item of backtrack, change nextver into last popped item in backtrack, and append the node into path.

```
            else:
                backtrack.pop()
                backtracked+=1
                try:
                    nextver = backtrack[-1]
                except:
                    print("Graph has a node that cannot be visited")
                    for i in range(backtracked-1):path.pop()
                    break

                if dijkstra_true:
                    path.pop()
                else:
                    path.append(nextver)

            if currentver == start:
                backtracked = 0
    return path
```

2. Kruskal's MST

```
# A pair is a list containing 3 values, [weight, source node, destination node]
# A value in a pstack is a list containing 2 values [node/vertex, weight]
```

```
def getPairs(graph, size):
    pairs = []
    for src in range(size):
        for dest in range(src, size):
            weight = graph[src][dest]
            if weight == inf:
                continue
            else:
                pairs.append([weight, src, dest])
    return pairs
```

The getPairs() function returns an array containing all the pairs in a given graph and accepts 2 parameters, the graph to be processed and the size of the graph (the amount of vertices the graph has).

```
def buildGraph(pairs, size):
    graph = [[inf for i in range(0,size)] for x in range(0,size)]

    for pair in pairs:
        graph[pair[1]][pair[2]] = pair[0]
        graph[pair[2]][pair[1]] = pair[0]

    return graph
```

The counterpart to getPairs(), this function returns a graph in a matrix form based on a given pair array. The graph is first initialized based on the size of the graph (the vertices) and all values will first be set as inf. The for loop then populates the matrix with the values from the pair array.

```
def kruskal(graph):
    original_graph = graph
    vertices = len(graph)
    k_pairs = []
    weight_pairs = getPairs(original_graph, vertices)

    weight_pairs.sort(key=lambda x: x[0])
```

Defines the kruskal function, which accepts a graph as its parameter. The initial variables are declared at the beginning of the function. The original_graph variable is to store the graph from the parameter, vertices is the amount of vertices in the graph, k_pairs is the array that will store valid pairs, weight_pairs is the array containing all the pairs from the original graph using the getPairs(original_graph, vertices) function. The getPairs() function turns a adjacent matrix into a pair array. Weight_pairs is then sorted based on index 0, which is the weight of the pair, which the top most pair in the stack will be the pair which has the lowest weight value.

```
disjoint_set = [[x] for x in range(vertices)]
location_set = list(range(vertices))
```

Initializes the disjoint_set and location_set, both variables are in the form of an array. The disjoint_set stores the sets of nodes that are already connected, when initialized each node will be in its own set. The location_set is to store the location of the nodes in the disjoint_set.

```
while len(disjoint_set) > 1:

    try:
        pair = weight_pairs.pop(0)
    except:
        print("Graph has a node that cannot be visited")
        break
```

If the disjoint set is greater than one, it means that not all possibly connected nodes are connected yet. For each iteration a pair is popped from the pair array, since the program will process that pair. The try except is to catch if the pair array is empty but the disjoint set is still greater than 1, meaning there are some disjoint sets that are disconnected from one another.

```
if not (pair[1] in disjoint_set[location_set[pair[2]]]):
    k_pairs.append(pair)

    disjoint_set[location_set[pair[1]]] = disjoint_set[location_set[pair[1]]] + disjoint_set[location_set[pair[2]]]
    disjoint_set.pop(location_set[pair[2]])

    for location in range(location_set[pair[2]], len(disjoint_set)):
        for x in disjoint_set[location]:
            location_set[x] = location_set[x] - 1

    for location in disjoint_set[location_set[pair[1]]]:
        location_set[location] = location_set[pair[1]]
```

```
if not (pair[1] in disjoint_set[location_set[pair[2]]]):
    k_pairs.append(pair)
```

Checks if the current source node is in the disjoint set of the destination node, if it is we do not want to add this pair to the k_pairs array, if it is not in the destination node's disjoint set, then the pair will be appended to the k_pair array.

```
disjoint_set[location_set[pair[1]]] = disjoint_set[location_set[pair[1]]] + disjoint_set[location_set[pair[2]]]
disjoint_set.pop(location_set[pair[2]])
```

Add the destination node to the disjoint set of the source node, meaning both node will have a connection, then pop or delete the disjoint set of the destination node, since it has been added to the disjoint set of source node.

```
for location in range(location_set[pair[2]], len(disjoint_set)):
    for x in disjoint_set[location]:
        location_set[x] = location_set[x] - 1
```

Since the disjoint set of the destination node has been popped, the locations (index) of all the disjoint sets to the right of it will need to be changed, since the len of the disjoint_set has also changed due to the pop. Every location to the left of the destination node in the disjoint_set array will need to be shifted to the left by one.

```
for location in disjoint_set[location_set[pair[1]]]:
    location_set[location] = location_set[pair[1]]
```

Since the disjoint set of the source and destination nodes in this pair has been merged, the location of the destination node's disjoint set will need to be updated. The location of the destination node's disjoint set will need to be updated to the location of the source node's disjoint set, since the destination node has been moved to the source node's disjoint set.

```
return buildGraph(k_pairs, vertices)
```

Once the program exits the loop it will return a matrix built by the buildGraph() function from the k_pairs, which contains all the valid pairs.

3. Dijkstra's SPT

```
def dijkstra(graph, start):
    not_visited = pstack()
    not_visited.append((start,0))
    vertices = len(graph)
    pairs = []
```

Defines the dijkstra() function; which accepts two parameters: graph and start (the starting node of the graph); and will return a SPT version of the given graph.
First the variables are initialized, not_visited is a pstack() object and is first appended with a tuple containing the starting node and the weight of that node (the starting node will always have zero as its weight), vertices is the amount of vertices in the graph or it's size, and the pairs array is first initialized as an empty array.

```
result = list(map(lambda x: 0 if x == start else inf, [x for x in range(vertices)]))
```

The result variable is a list that is first initialized with the length equal to the amount of vertices of the graph, this list represents the path value of every node. This list is first initialized to have all values be inifinity except for the starting node's path value, which will always be zero. This simulates how at the beginning of Dijkstra's algorithm the only known path is to the starting node, and the path to all other nodes are still unknown, which is set as infinity.

```
while not not_visited.isEmpty():

    current_node = not_visited.pop()

    for edge in range(vertices):

        if graph[current_node[0]][edge] == inf:
            continue

        current_weight = current_node[1] + graph[current_node[0]][edge]

        if current_weight < result[edge]:

            result[edge] = current_weight
            not_visited.append((edge, current_weight))

            for pair in range(len(pairs)):
                if edge == pairs[pair][1] or edge == pairs[pair][2]:
                    pairs.pop(pair)
                    break
            pairs.append([graph[current_node[0]][edge], edge, current_node[0]])
        else:
            continue
return buildGraph(pairs, vertices)
```

```
while not not_visited.isEmpty():
```

The while loop is executed as long as there are still unvisited nodes.

```
current_node = not_visited.pop()
```

Gets the current working node

```
for edge in range(vertices):

    if graph[current_node[0]][edge] == inf:
        continue

    current_weight = current_node[1] + graph[current_node[0]][edge]

    if current_weight < result[edge]:

        result[edge] = current_weight
        not_visited.append((edge, current_weight))

        for pair in range(len(pairs)):
            if edge == pairs[pair][1] or edge == pairs[pair][2]:
                pairs.pop(pair)
                break
        pairs.append([graph[current_node[0]][edge], edge, current_node[0]])
    else:
        continue
```

Checks all the values of the nodes connected to the current working node.

```
if graph[current_node[0]][edge] == inf:
    continue
```

If the value of the connected node is infinity, go to the next connected node

```
current_weight = current_node[1] + graph[current_node[0]][edge]
```

Declares the weight of the current node, which is the sum of the source node and the connected node.

```
if current_weight < result[edge]:

    result[edge] = current_weight
    not_visited.append((edge, current_weight))

    for pair in range(len(pairs)):
        if edge == pairs[pair][1] or edge == pairs[pair][2]:
            pairs.pop(pair)
            break
    pairs.append([graph[current_node[0]][edge], edge, current_node[0]])
else:
    continue
```

If the weight of the current node is less than the weight in the result array, than we update the result array. We set the weight of the current node in the result array with the current_weight. Then the connected node is added to the not visited pstack() along with its current_weight. The pair that has the connected node is then popped from the pair list, since a new pair with a smaller weight value is going to be appended. Once the old pair has been popped the new pair is appended to the pair array.
If the current_weight is greater than the weight in the result array, this pair will not be added to the pair list, thus will enter the else statement and continue the loop and proceed to the next connected node.

```
return buildGraph(pairs, vertices)
```

Once all the pairs have been processed the matrix will be rebuilt using the buldGraph() function and is returned.

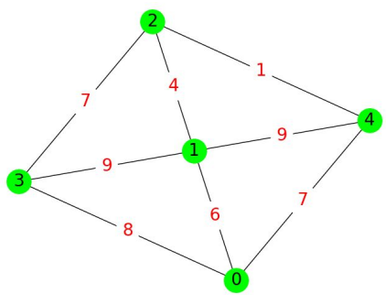
- Data sets

  The data sets for the program are graphs in form of adjacency matrix. The graphs are generated by the size of user's input, if the size of the graph is five then there will be five nodes in the graph with each five possible paths. The values inside the matrix are randomly generated, if zero is generated then it means that the node does not have a path to its index. All zeroes will change value into infinite values.

  We will use the second preset matrix from the code to plot the graph and analyse it.

```
Graph 2
Node 0:     [inf, 6, inf, 8, 7]
Node 1:     [6, inf, 4, 9, 9]
Node 2:     [inf, 4, inf, 7, 1]
Node 3:     [8, 9, 7, inf, inf]
Node 4:     [7, 9, 1, inf, inf]
```

Adjacency matrix



Plotted graph

- Experimental analysis

  To analyse the graphs, we test it on an unmodified graph, MST Kruskal modified graph, and SPT Dijkstra modified graph.

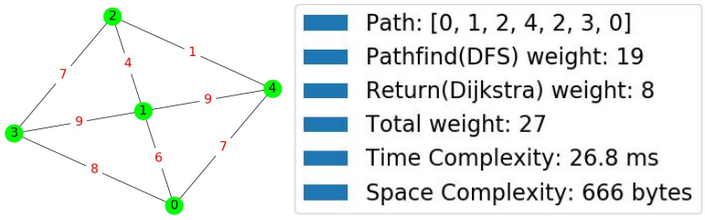
  - **Kruskal:** Searches the smallest path/weight in a graph, then put the two connected nodes into the same disjoint set. This is to check for looping. Repeats the above process until all the nodes are in the same disjoint set. Once all nodes are in the same disjoint set it would mean that all the nodes are connected in some way
  - **Dijkstra:** Requires a given starting point. Generates a Shortest Path Tree, which will always give the shortest path from the given starting point to any node.

    How Dijkstra's Algorithm works

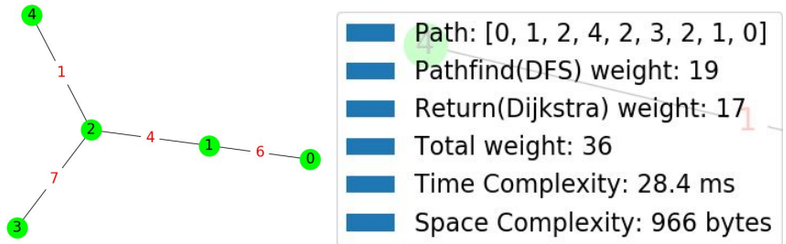
    1. All the values of nodes (the weight of the node path from the starting point) not directly connected to the starting node will be set as infinity
    2. At first the starting node will be the current node
    3. The next node will be the node connected to the current node with the smallest value
    4. Check if all nodes connected to the next node
       a. For each connected node, check if the value sum of next node and the connected node is smaller than the current value of that connected node.
       b. If the value is smaller, than update the value of the connected node with the sum
    5. Set next node as the current node than go to step 3
    6. Repeat steps 3 to 5 until all nodes are visited
  - **DFS:** Traverses through all the nodes in the graph. DFS tries to reach the "deepest" part of a branch then backtracks to the nearest node that has a different path; this is an important aspect for our pathfinding algorithm.

- **Path**: The path from post office to all addresses, and return path from the last address back to the post office.
- **Pathfind(DFS) weight**: Total graph weight to visit all nodes.
- **Return(Dijkstra) weight**: Total graph weight to return to node zero from the last node in pathfind function.
- **Total weight**: Sum of pathfind and return weight
- **Time Complexity**: Counts how many time each line of code runs while running each function.
- **Space Complexity**: Counts how much memory size each function uses while running, by getting the size of each data type in the function.

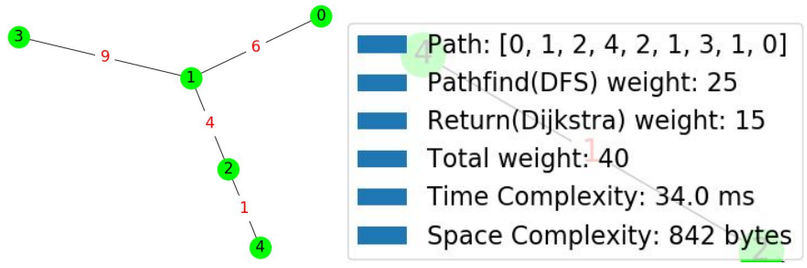
- **Result**
  1. Unmodified graph



  2. MST Kruskal modified graph



  3. SPT Dijkstra modified graph



5. Discussion

We chose DFS for our traversal algorithm, since we found it to be the most suitable for our needs. We chose DFS because of its ability to backtrack. We implement DFS to find the path for the mail courier to reach all the destinations.

We chose Dijkstra's algorithm for producing our SPT. We used Dijkstra's algorithm since it was enough for our needs, it allowed us to find the shortest path from one node to the other.

For producing our MST we chose to use Kruskal's algorithm, it was a simple algorithm that we could implement on our program. We chose Kruskal's algorithm instead of Prim's algorithm since we found that we could implement Kruskal's algorithm in a similar way to our implementation of Dijkstra's algorithm.

There are still some parts of the code we could improve and maybe add more feature at the end, but concerned with the time we settled on the current

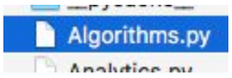
program features and fix any bugs that are still present rigorously test our program for any bugs we might have missed.

6. Conclusion and Recommendation

The result may vary depending on the graph. The MST version can sometimes produce a more inefficient route compared to using the original unmodified graph. The SPT version requires manual input of starting point so it is tedious to use, however the use of Dijkstra's algorithm is mainly to calculate the return trip. It is recommended to try and compare the unmodified and MST version while using the program.

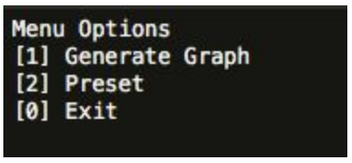
7. Program Manual

First run the Algorithms.py

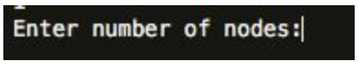


(No need to run Analytics.py, it is just a module used by the main Algorithms.py)

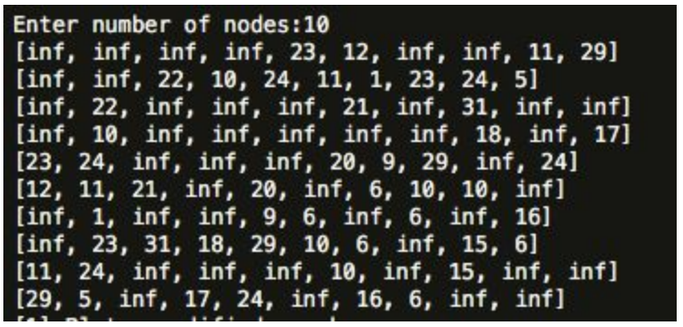
When the program is executed there will be a menu that will prompt the user to select whether to select a preset graph or generate a random graph



If the user decides to generate a random graph, a prompt to input the size of the graph (the amount of nodes in the graph)



The program will generate a graph based on the size given by the user



The graph generated will be in the form of an adjacency matrix

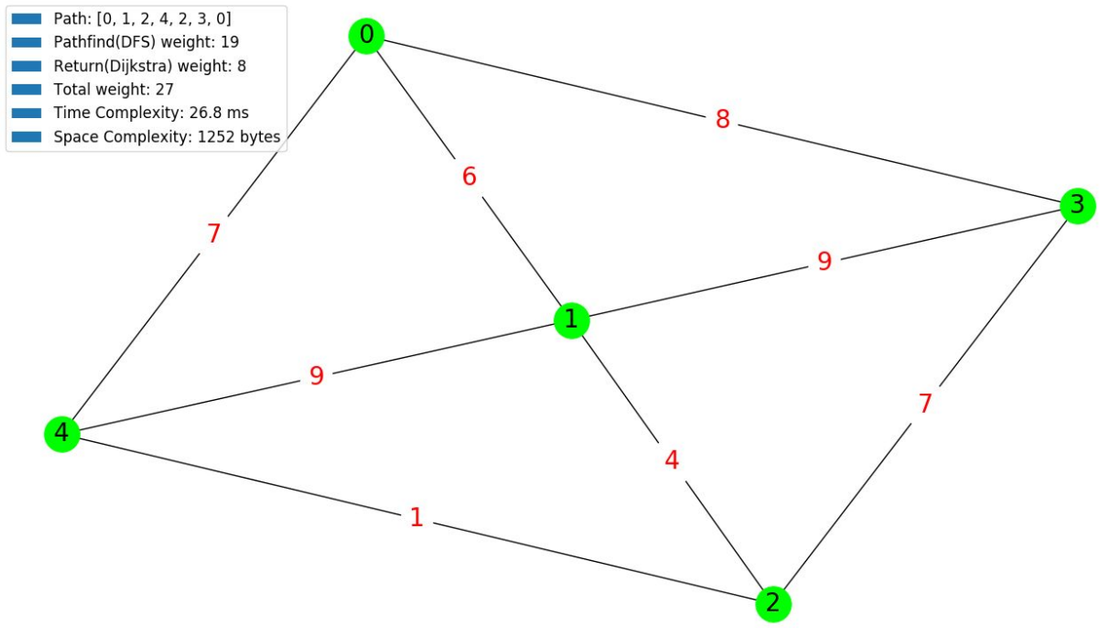
If the user decides to use a preset graph, the preset graphs will be displayed and the user will be prompted to select a graph

```
Availilable Graphs
Graph 1
Node 0:     [inf, inf, 6, 2, 3]
Node 1:     [inf, inf, 1, 6, 2]
Node 2:     [6, 1, inf, 1, 9]
Node 3:     [2, 6, 1, inf, 1]
Node 4:     [3, 2, 9, 1, inf]
Graph 2
Node 0:     [inf, 6, inf, 8, 7]
Node 1:     [6, inf, 4, 9, 9]
Node 2:     [inf, 4, inf, 7, 1]
Node 3:     [8, 9, 7, inf, inf]
Node 4:     [7, 9, 1, inf, inf]
Graph 3
Node 0:     [inf, 6, 8, inf, inf]
Node 1:     [6, inf, 9, 8, inf]
Node 2:     [8, 9, inf, 8, inf]
Node 3:     [inf, 8, 8, inf, inf]
Node 4:     [inf, inf, inf, inf, inf]
Graph 4
Node 0:      [inf, 5, inf, 3, 5, 8]
Node 1:      [5, inf, 3, 4, inf, inf]
Node 2:      [inf, 3, inf, inf, 6, 2]
Node 3:      [3, 4, inf, inf, 2, inf]
Node 4:      [5, inf, 6, 2, inf, inf]
Node 5:      [8, inf, 2, inf, inf, inf]
Please select a Graph
[1] Graph 1
[2] Graph 2
[3] Graph 3
[4] Graph 4
[0] Exit
```

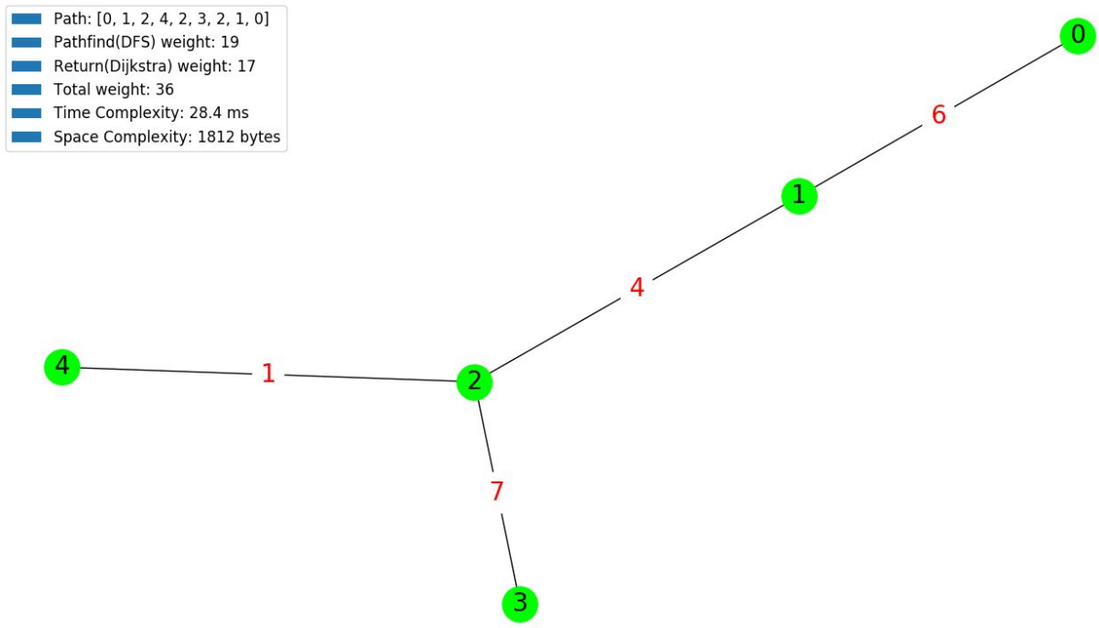
Once a graph has been selected the user will be prompted to select the options above

```
[1] Plot unmodified graph
[2] Plot MST (Kruskal) graph
[3] Plot SPT (Dijkstra) graph
[4] Select different graph
[0] Exit
```

Option no 1 plots the original unmodified graph and displays the information on the plot (example uses preset graph 2)



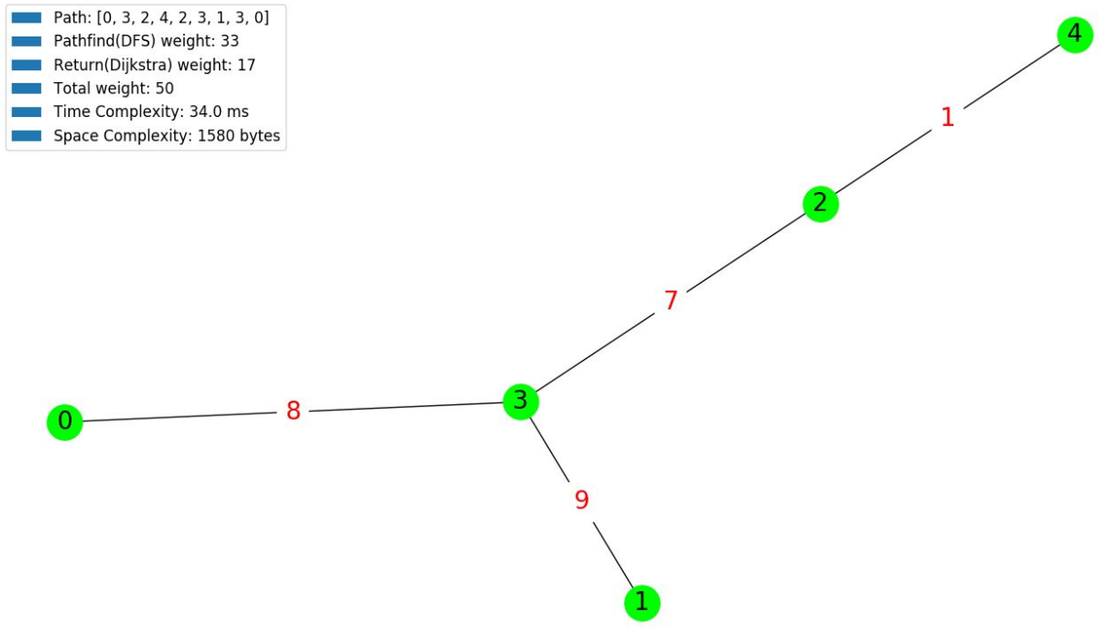
Option no 2 plots the MST graph and displays the information on the plot (example uses preset graph 2)



Option no 3 will prompt the user to input a starting point

```
3
Please input a starting point
```

The program will then plot an SPT graph based on the starting point given by the user (example uses preset graph 2, starting point 3)



Option no 4 is for the user to select a different graph

```
Menu Options
[1] Generate Graph
[2] Preset
[0] Exit
```

Option no 0 is to exit the program

```
4
Menu Options
[1] Generate Graph
[2] Preset
[0] Exit
0

Run Succeeded   Time 0:00:21
```

Video Link :
https://drive.google.com/open?id=1Mb9peHLvs9Pyhghamu7Y0gkejfJDuFan

GitHub Link :
https://github.com/fiqhyb/Analysis_Of_Algorithm