Introduction

This documentation will explain about the **two** problems chosen and provide a full elaboration about it. For the first problem, we have chosen the Binary Search Tree whereas the data structure is also called a binary tree, that has properties of arranging nodes in a specific order. like the value of all the nodes on the left subtree are lesser than the root node, while the value of the nodes on the right are greater than the root node.

The next problem chosen is the Dijkstra Algorithm, which is also a form of greedy algorithm that finds the shortest path from a starting vertex to all the other vertices. The graph data structure is used for this instance, which contains vertices that represent an object and the edges that connects in between vertices. For this scenario, the graph is undirected as any nodes that are connected can go from both directions and depending on either it is a weighted or non-weighted graph, the shortest path would be different too.

Additionally, the tools used to code the algorithms is the Spyder IDE for Python language, as it contains components that helps in coding like variable explorer and easily integrate / import different functions that makes coding much easier. As mentioned, the programming language used is Python, the default features are used commonly to implement Object-Oriented Programming methods. Moreover, a few imported modules were added to support the code with its features, like the Heapq module that helps with priority queueing in the Dijkstra Algorithm. The core feature is the NetworkX Python package, that helps with providing functions that creates and manipulates the graph, while it utilizes Pyplot from Matplotlib to allow visualization of the graph itself.

Problem 1 Implementation – Binary Search Tree

Explanation

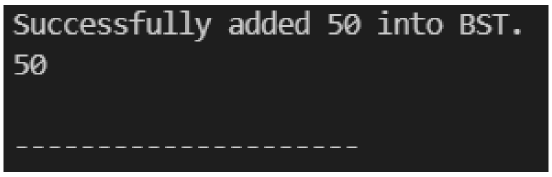
Binary Search Tree (BST) is used to store data in an organized way. It is a recursive data structure where each node can have only 2 children. The value of all the nodes in the left sub-tree is less than the value of the root while value of all the nodes in the right sub-tree is greater or equal to the value of the root. Binary tree is useful for stored data to be fetched, inserted, updated, and deleted quickly. We will be also discussing additional features that we add such as traversal, getting minimum and maximum and visualisation.

Demonstration of Insertion of node

In this implementation, there are a few different functions that can be used with the BST. However, the main function is inserting a node into the tree. For each value being inserted into the BST, there are three rules to be followed. For easier explanation, a demonstration is used to better understand how the rules work:

Rule 1: If there is no value or data in the BST yet, the first number entered to the program will be automatically assigned as root. We will take the number '50' as an example.

Integer 50 being inserted into BST:

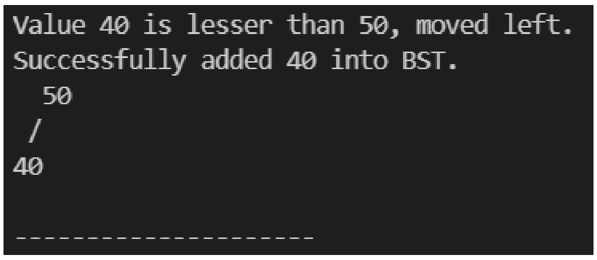


```
Successfully added 50 into BST.
50

----------------------
```

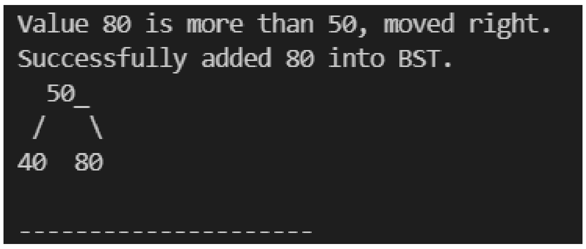
50 is now inserted as the root of BST.

Rule 2: If the value to be inserted is lesser than the current value, it will be assigned to the left child of the root. If the number is more than the root number, it will be assigned to the right child of the root.

Integer 40 being inserted into BST:

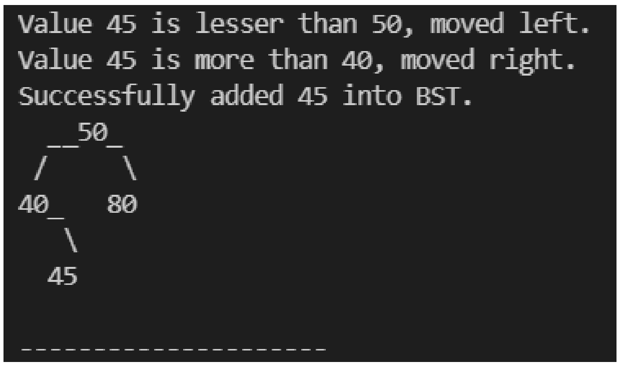

```
Value 40 is lesser than 50, moved left.
Successfully added 40 into BST.
  50
 /
40

--------------------
```
Since 40 is less than 50, it will be assigned to left child.

Integer 80 being inserted into BST:


```
Value 80 is more than 50, moved right.
Successfully added 80 into BST.
  50_
 /   \
40  80

--------------------
```
Since 80 is larger than 50, it will be assigned to right child.

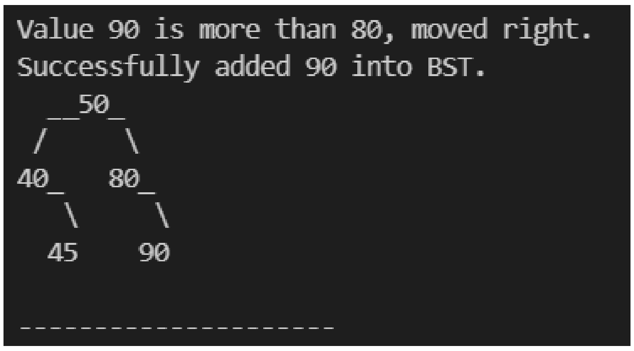
Rule 3: For all of the subsequent integers being added into the BST, it will continuously be inserted below the sub-tree and follows the same rules as before. However, it is important to note that each node can only have **one left child** and **one right child**. If a node has a left child and right child already, it is considered full and the new values have to be assigned as children to the child of the node instead.

Integer 45 being inserted into BST:


```
Value 45 is lesser than 50, moved left.
Value 45 is more than 40, moved right.
Successfully added 45 into BST.
  __50_
 /     \
40_   80
   \
  45

--------------------
```

Since 45 is lesser than 50 but there already is a left child node for 50, 45 is compared to the value of the left child node, which is 40, and placed accordingly.

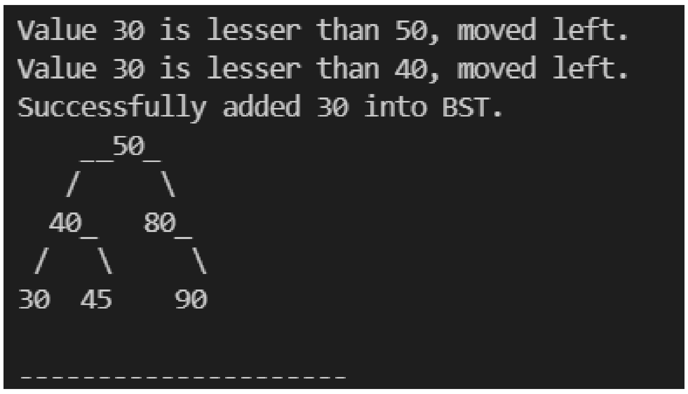
Integer 90 being inserted into BST:



```
Value 90 is more than 80, moved right.
Successfully added 90 into BST.
  __50_
 /     \
40_   80_
   \     \
  45    90

---------------------
```

Since 90 is larger than 50 but there already is a right child node for 50, 90 is compared to the value of the right child node, which is 80, and placed accordingly.

Integer 30 being inserted into BST:

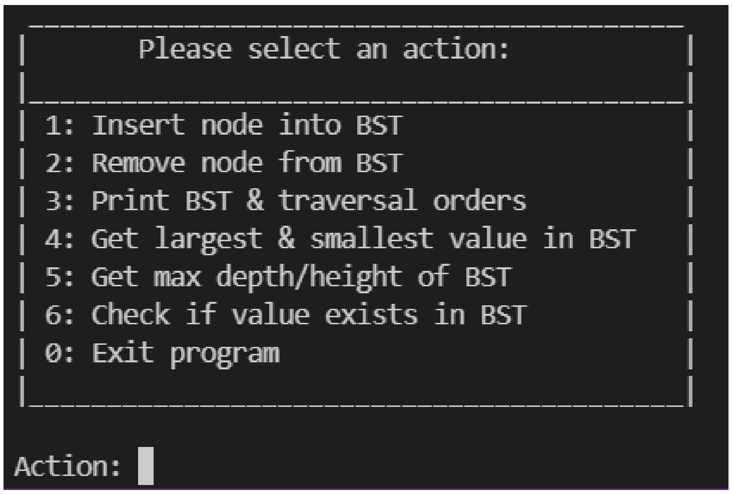


```
Value 30 is lesser than 50, moved left.
Value 30 is lesser than 40, moved left.
Successfully added 30 into BST.
    __50_
   /     \
  40_   80_
 /   \     \
30  45    90

---------------------
```

The same concept goes for 30, as well as all other new values to be added into the BST.

Functions in the Implementation

- **Print Menu options**



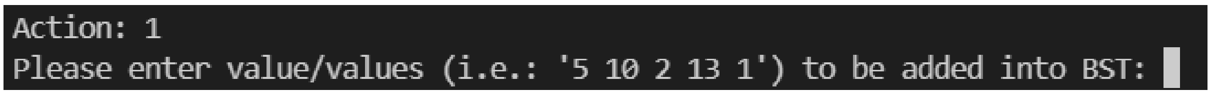
```
 ________________________________________
|        Please select an action:        |
|________________________________________|
| 1: Insert node into BST                |
| 2: Remove node from BST                |
| 3: Print BST & traversal orders        |
| 4: Get largest & smallest value in BST |
| 5: Get max depth/height of BST         |
| 6: Check if value exists in BST        |
| 0: Exit program                        |
|________________________________________|

Action:
```

This function acts as a medium for the user to conveniently interact with the BST and perform all of the functions. The user is supposed to enter a number based on which action they would like to perform. There is also error handling if a user enters an invalid input.

---

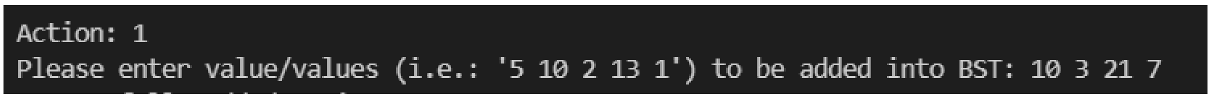
1. Insert node into BST

As discussed earlier, this function adds a new element to the binary search tree at the proper location. This function accepts single values as well as multiple values. During each insertion, the value is compared with other nodes and the updated BST will be printed out in the terminal.



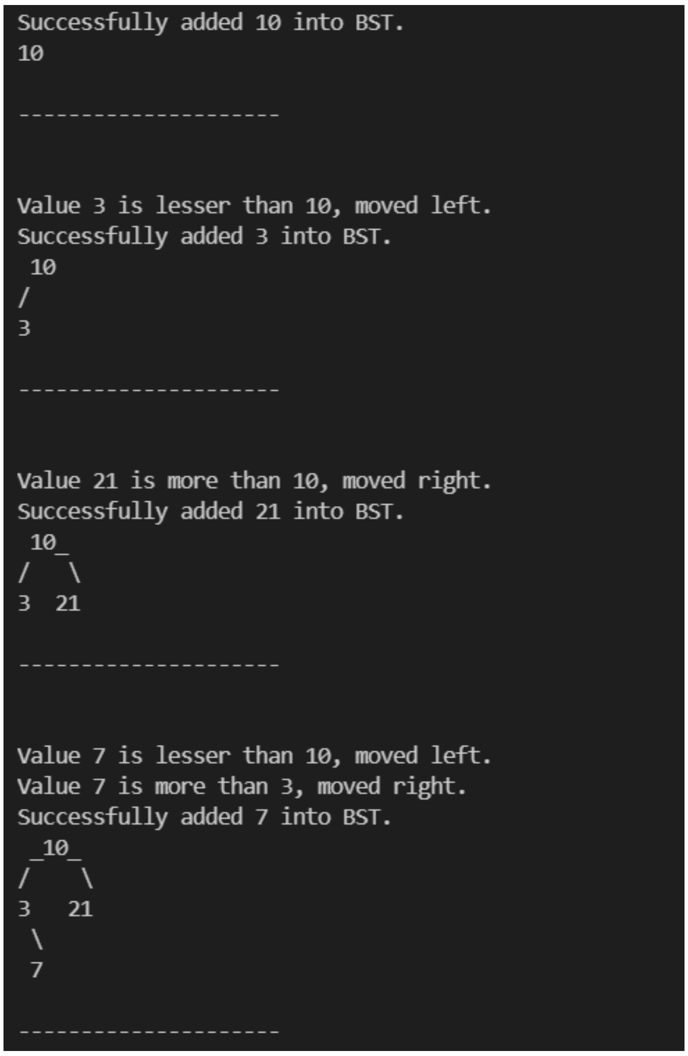
```
Action: 1
Please enter value/values (i.e.: '5 10 2 13 1') to be added into BST:
```

For example, the values 10, 3, 21, and 7 is inserted into the BST:



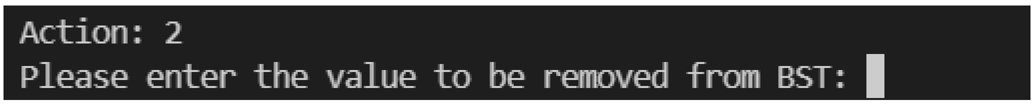
```
Action: 1
Please enter value/values (i.e.: '5 10 2 13 1') to be added into BST: 10 3 21 7
```

Resulted output:

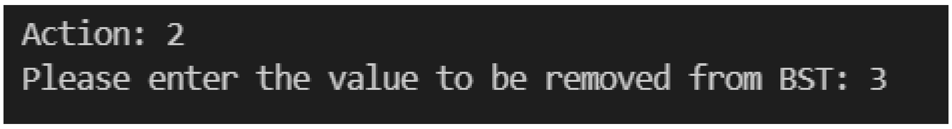

```
Successfully added 10 into BST.
10

--------------------


Value 3 is lesser than 10, moved left.
Successfully added 3 into BST.
 10
/
3

--------------------


Value 21 is more than 10, moved right.
Successfully added 21 into BST.
 10_
/   \
3  21

--------------------


Value 7 is lesser than 10, moved left.
Value 7 is more than 3, moved right.
Successfully added 7 into BST.
 _10_
/    \
3   21
 \
 7

--------------------
```

2. Remove node from BST

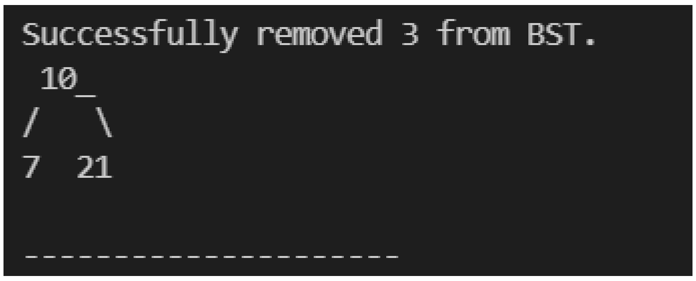
This function allows for the removal of a node in the BST. The program will prompt the user to enter the value to be removed. Then, it will print the updated BST with the removed node.


```
Action: 2
Please enter the value to be removed from BST: 
```

As an example, the BST created above is taken. From the existing tree above with values 10, 3, 21, and 7, we would like to remove 3 from the BST:

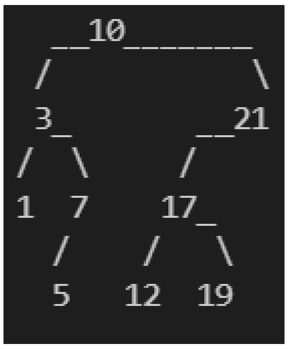

```
Action: 2
Please enter the value to be removed from BST: 3
```

Resulted output:



```
Successfully removed 3 from BST.
 10_
/   \
7  21


---------------------
```

****For the functions from 3 to 6, this is the example tree that is going to be used:**



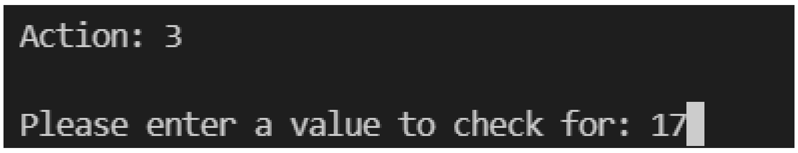
```
  __10_______
 /           \
 3_        __21
/  \      /
1  7    17_
  /    /   \
  5   12  19
```

---

3. Check if value exists in BST

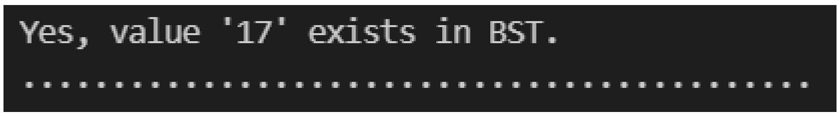
A value inserted into a BST can be searched by recursively iterating through all of its nodes until it is found. This function utilizes the same methodology to display whether a value is already inside of a BST.

To check if the value 17 exists in the BST:



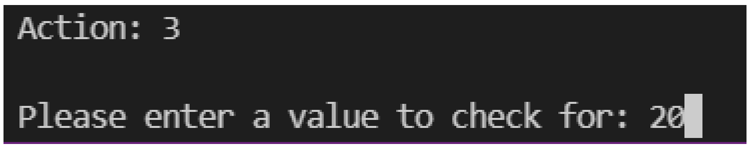
```
Action: 3

Please enter a value to check for: 17
```

Resulted output:



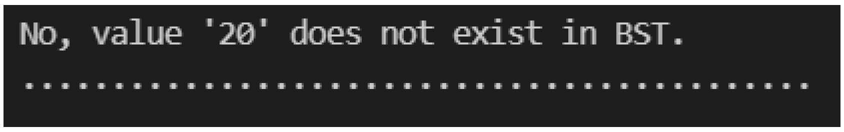
```
Yes, value '17' exists in BST.
............................................
```

To check if value 20 exists in the BST:



```
Action: 3

Please enter a value to check for: 20
```

Resulted output:

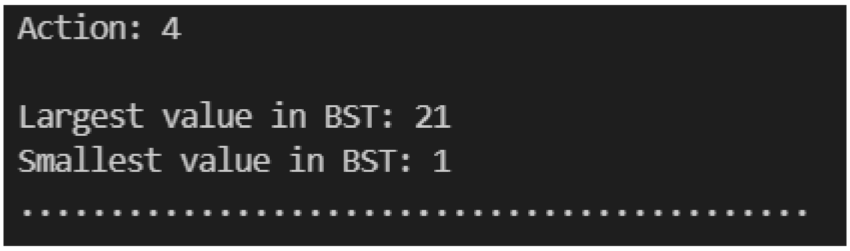


```
No, value '20' does not exist in BST.
.............................................
```

---

4. Get largest & smallest value in BST

The largest and smallest value in the BST will be displayed. This function utilizes the inorder traversal to obtain both values. This is because the inorder traversal sorts the values by ascending order. This means that the smallest value is always at the first element while the largest value is the last element in the inorder traversal.

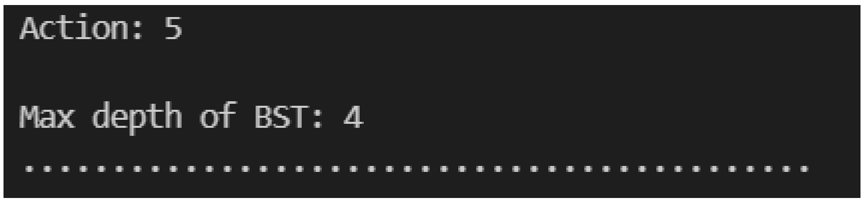


```
Action: 4

Largest value in BST: 21
Smallest value in BST: 1
.............................................
```

---

5. Get max depth/height of BST

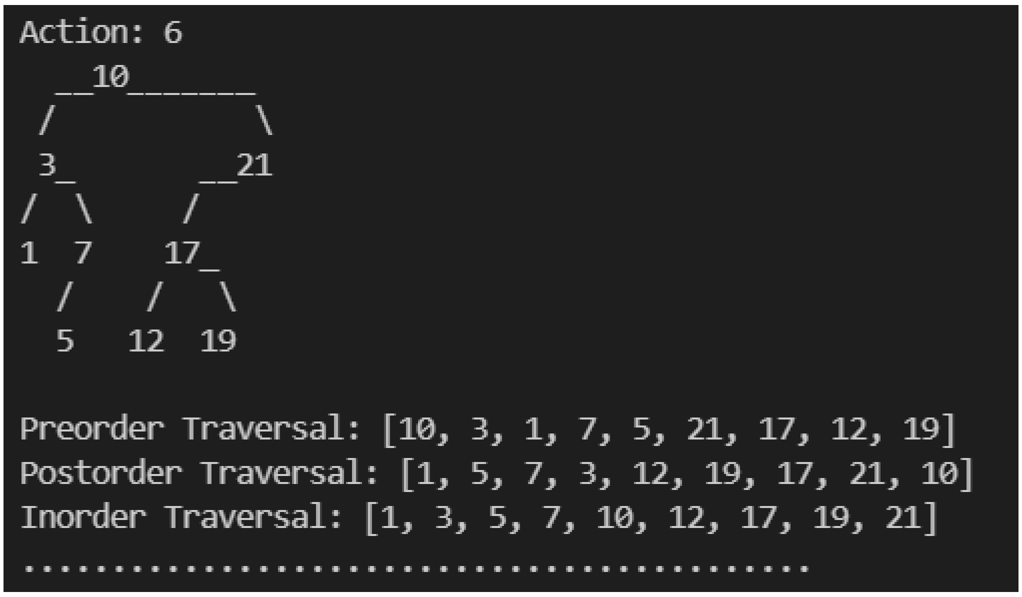
A BST with at least one value has a height. The height can be interpreted as how many levels the BST has. To obtain the height, a recursive function is used. To simplify, for each child node that the recursive function passes through, the height will increase by 1. When the functions reaches the leaf node, the height of the tree is returned.



```
Action: 5

Max depth of BST: 4
.............................................
```

6. Print BST & traversal orders

This function has two features: visualizing the BST as well as its traversal orders. The visualization part displays the current BST. For the traversal orders, there are preorder traversal, postorder traversal, and inorder traversal.

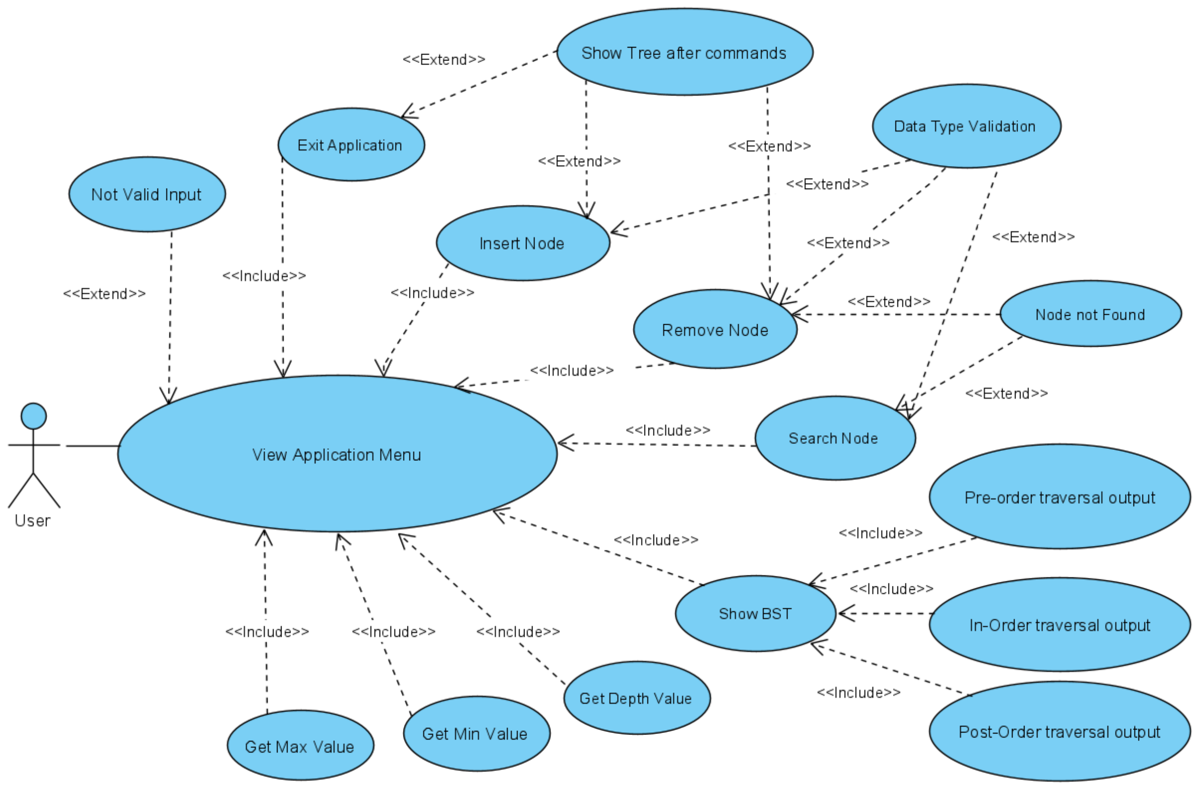


```
Action: 6
  __10_______
 /           \
 3_        __21
/  \      /
1  7    17_
  /    /   \
  5   12  19

Preorder Traversal: [10, 3, 1, 7, 5, 21, 17, 12, 19]
Postorder Traversal: [1, 5, 7, 3, 12, 19, 17, 21, 10]
Inorder Traversal: [1, 3, 5, 7, 10, 12, 17, 19, 21]
...........................................
```

The inorder traversal also plays an important role in function 4 because with inorder traversal, the data is sorted by ascending order. Therefore, the minimum and maximum value can be obtained from it.

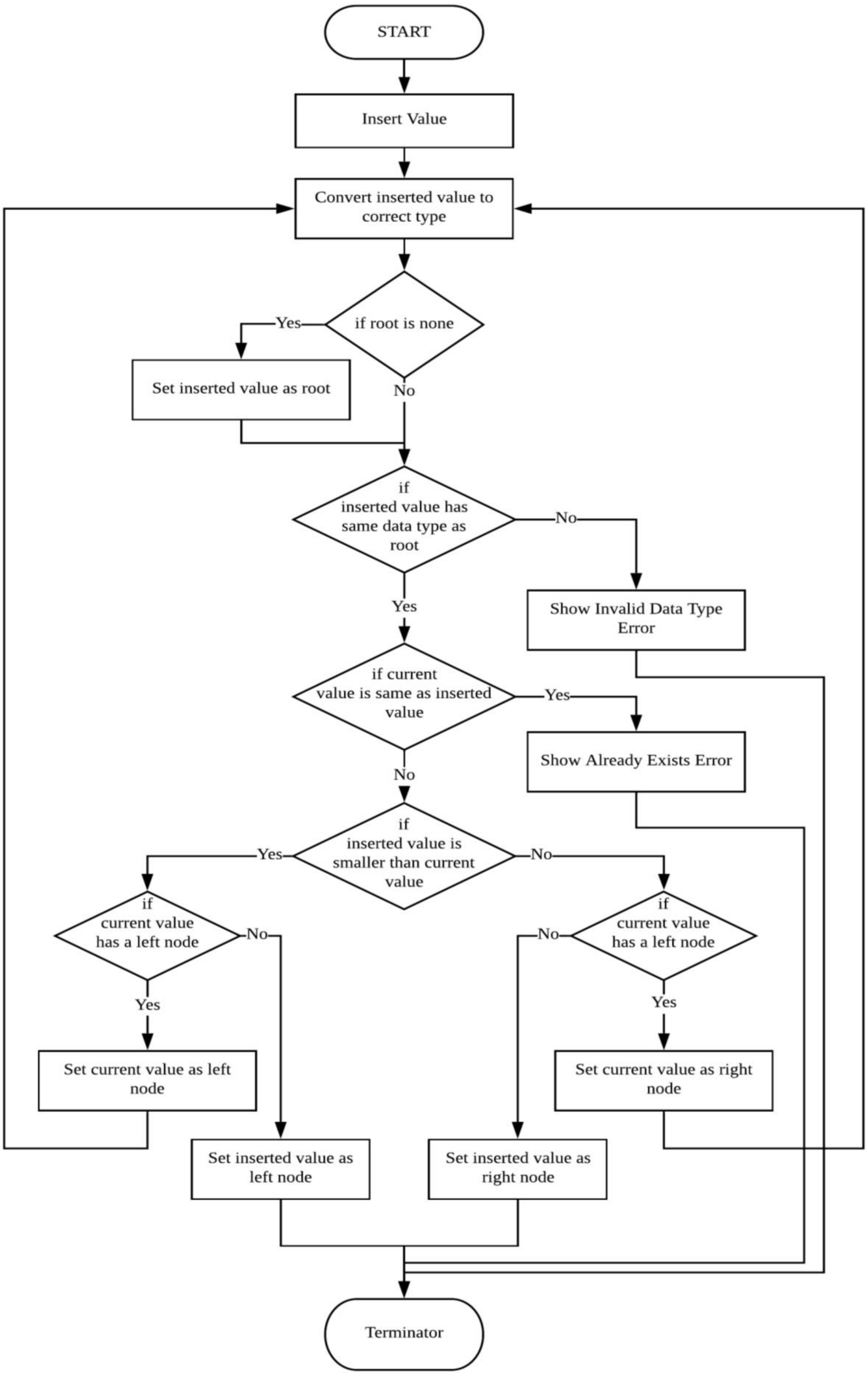
Diagrams for the Implementation

1. UML Use Case Diagram for BST Implementation



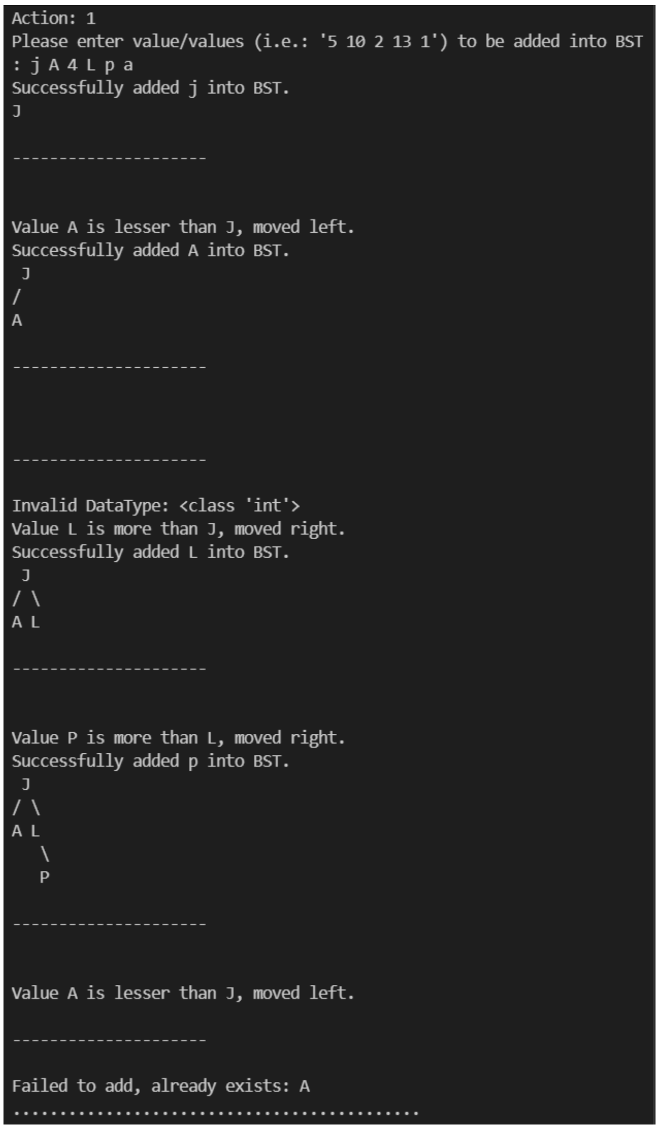
2. Flowchart for BST Implementation

The Insertion of Node

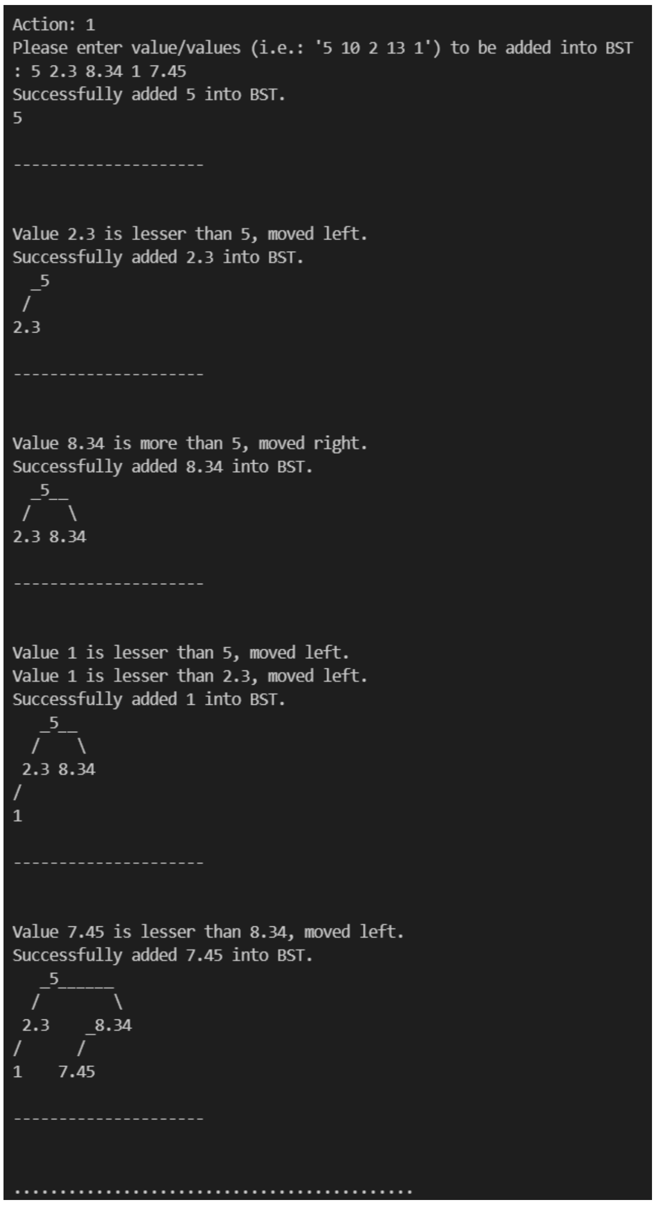

START
Insert Value
Convert inserted value to correct type
if root is none
Yes
Set inserted value as root
No
if inserted value has same data type as root
No
Show Invalid Data Type Error
Yes
if current value is same as inserted value
Yes
Show Already Exists Error
No
if inserted value is smaller than current value
Yes
No
if current value has a left node
No
Yes
Set current value as left node
Set inserted value as left node
if current value has a left node
No
Yes
Set current value as right node
Set inserted value as right node
Terminator

Additional Screenshots of the Implementation

The following demonstrates the output for error handling as well as supporting various data types:

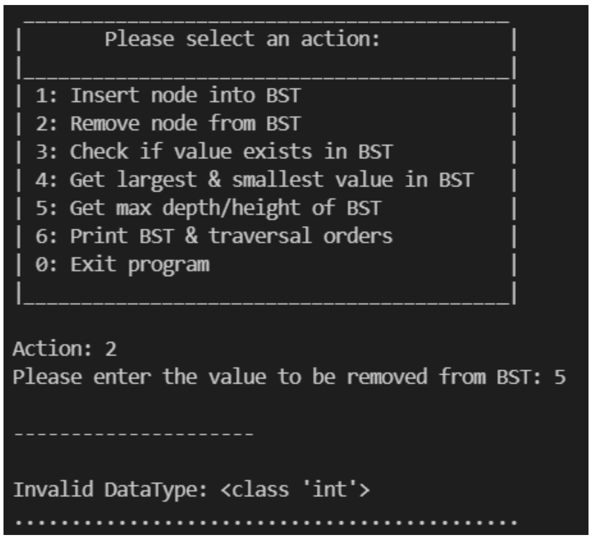


```
Action: 1
Please enter value/values (i.e.: '5 10 2 13 1') to be added into BST
: j A 4 L p a
Successfully added j into BST.
J

--------------------


Value A is lesser than J, moved left.
Successfully added A into BST.
 J
/
A

--------------------




--------------------

Invalid DataType: <class 'int'>
Value L is more than J, moved right.
Successfully added L into BST.
 J
/ \
A L

--------------------


Value P is more than L, moved right.
Successfully added p into BST.
 J
/ \
A L
   \
   P

--------------------


Value A is lesser than J, moved left.

--------------------

Failed to add, already exists: A
............................................
```



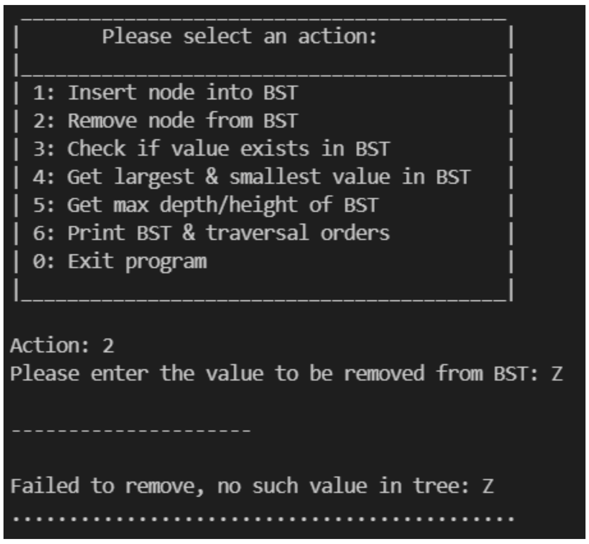
```
Action: 1
Please enter value/values (i.e.: '5 10 2 13 1') to be added into BST
: 5 2.3 8.34 1 7.45
Successfully added 5 into BST.
5

--------------------


Value 2.3 is lesser than 5, moved left.
Successfully added 2.3 into BST.
  _5
 /
2.3

--------------------


Value 8.34 is more than 5, moved right.
Successfully added 8.34 into BST.
  _5__
 /    \
2.3 8.34

--------------------


Value 1 is lesser than 5, moved left.
Value 1 is lesser than 2.3, moved left.
Successfully added 1 into BST.
   _5__
  /    \
 2.3 8.34
/
1

--------------------


Value 7.45 is lesser than 8.34, moved left.
Successfully added 7.45 into BST.
   _5______
  /        \
 2.3     _8.34
/       /
1    7.45

--------------------



............................................
```

Data type does not match data type of root:



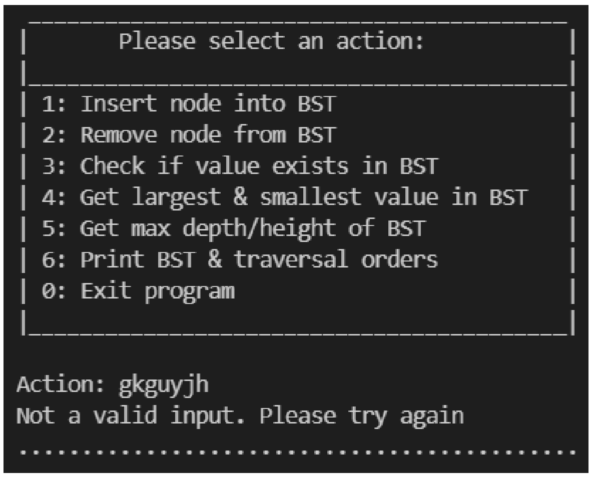
```
 ________________________________________
|       Please select an action:         |
|________________________________________|
| 1: Insert node into BST                |
| 2: Remove node from BST                |
| 3: Check if value exists in BST        |
| 4: Get largest & smallest value in BST |
| 5: Get max depth/height of BST         |
| 6: Print BST & traversal orders        |
| 0: Exit program                        |
|________________________________________|

Action: 2
Please enter the value to be removed from BST: 5

--------------------

Invalid DataType: <class 'int'>
.........................................
```

No such value in BST to be removed:



```
 ________________________________________
|       Please select an action:         |
|________________________________________|
| 1: Insert node into BST                |
| 2: Remove node from BST                |
| 3: Check if value exists in BST        |
| 4: Get largest & smallest value in BST |
| 5: Get max depth/height of BST         |
| 6: Print BST & traversal orders        |
| 0: Exit program                        |
|________________________________________|

Action: 2
Please enter the value to be removed from BST: Z

--------------------

Failed to remove, no such value in tree: Z
.........................................
```

Invalid action selected:



```
 ________________________________________
|       Please select an action:         |
|________________________________________|
| 1: Insert node into BST                |
| 2: Remove node from BST                |
| 3: Check if value exists in BST        |
| 4: Get largest & smallest value in BST |
| 5: Get max depth/height of BST         |
| 6: Print BST & traversal orders        |
| 0: Exit program                        |
|________________________________________|

Action: gkguyjh
Not a valid input. Please try again
.........................................
```

Additional Features

Support & validation for multiple data types

Besides the main functions, this implementation also supports BST of various data types, namely integer, float, and alphabetical. The system also validates the type of value being inserted into the BST. How this works is that the code validates the first value that is inserted as root node. After that, any data that is entered is then compared with the type of root node. For example, if the first value of the BST is an integer, this means that only integer or float can be inserted. If the user tries entering a string, the system will deny the request. This also works for string. If the root node is a string, the subsequent values can only be limited to string data type. This is done to ensure the consistency of data in the BST.

Visualization

For the visualization of the BST, the function that is responsible is credited to StackOverflow user 'J. V.' (https://stackoverflow.com/a/54074933). The function is able to print the BST into the terminal by utilizing underscores and slashes. It increase user friendliness of the implementation because the user can easily look at the arrangement of the tree and all of its values.

Limitations

High memory usage

Binary search tree can take more memory when compared to an array since entire objects are typically involved. And, when compared to other algorithm for brute-force search it is quite slow.

No proper Graphical User Interface

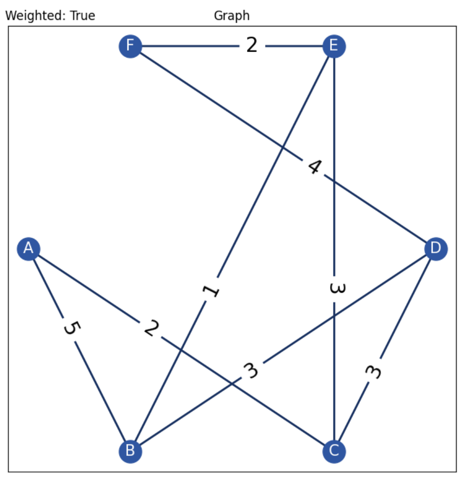
All of the functions of the BST involves the use of a console/terminal. There is no convenient or aesthetic approach for the user to interact with the BST without a terminal. Therefore, this implementation is not visually appealing.

Problem 2 Implementation – Dijkstra's Algorithm

Explanation

In a graph data structure, Dijkstra's algorithm is used to determine the shortest path from one node to another. The implementation of this algorithm can be used on weighted graphs as well as unweighted graphs. Besides that, this algorithm also works on directed and undirected graph. The graph for this implementation supports various data types, which are explained in detail in the "Additional Features" section. By default, the graph created will be weighted and undirected. These settings can be modified using a Boolean in the parameter for the function when creating the graph. For a convenient explanation of the Dijkstra function, a step-by-step example of the execution is used instead.

Demonstration of Dijkstra's Algorithm execution



The figure above shows a weighted, undirected graph with 6 nodes. The goal for this example is to find the shortest path from node 'A' to node 'F' using Dijkstra's algorithm.

Two dictionaries will be created to store and continuously update the current known shortest path from starting node 'A' to every other node. During initialization, the Distance dictionary and Path dictionary will have these values:

Distance dictionary	{ 'A': 0 , 'B': inf , 'C': inf , 'D': inf , 'E': inf , 'F': inf }
Path dictionary	{ 'A': ['A'] , 'B': ['A'] , 'C': ['A'] , 'D': ['A'] , 'E': ['A'] , 'F': ['A'] }

Then, a priority queue is created and will be utilized together with Python's 'heapq' module for push and pop functionalities. The function of the priority queue is to store and sort by the least distance to determine which node should be explored next. The priority queue contains the distance followed by the node. The initial value of the priority queue will be a distance of 0 followed by the starting node.

Priority queue	[(0 , 'A')]

After this, a 'while' loop is used. It will stop looping when the priority queue does not have any elements left. This indicates that the shortest possible path is already discovered.

While loop:

When an iteration starts, the first element of the priority queue will be popped and will be the focus node for this iteration. Then, a comparison is made. If the distance to the focus node is larger than the distance in the dictionary, this means that the algorithm has already found a shorter pathway. Therefore, the algorithm **skips the rest of the steps** and moves on to the second iteration. If the comparison returned false, all of the **neighbour nodes** for 'A' will be compared one by one using a 'for' loop.

For loop:

In this case, 'B' is compared first. The two values to be compared is the **total distance from starting node to the neighbour node** and the **currently known shortest distance to the neighbour node**. In the first iteration of the 'for' loop, the currently known shortest distance to the neighbour node 'B' is **infinity**. Therefore, the infinity value will be replaced with '5' for node 'B' and both dictionaries will be updated.

Distance dictionary	{ 'A': 0 , 'B': 5 , 'C': inf , 'D': inf , 'E': inf , 'F': inf }
Path dictionary	{ 'A': ['A'] , 'B': ['A','B'] , 'C': ['A'] , 'D': ['A'] , 'E': ['A'] , 'F': ['A'] }

Since a shorter path has been found, the values are pushed into the priority queue and the first iteration of the loop is finished. The new priority queue:

Priority queue	[(5 , 'B')]

For the second iteration of the ‘for’ loop, the following neighbour node is taken. For this case, it is node ‘C’. Since the distance from ‘A’ to ‘C’ is lesser than infinity, the dictionaries will be updated.

Distance dictionary	{ 'A': 0 , 'B': 5 , 'C': 2 , 'D': inf , 'E': inf , 'F': inf }
Path dictionary	{ 'A': ['A'] , 'B': ['A','B'] , 'C': ['A','C'] , 'D': ['A'] , 'E': ['A'] , 'F': ['A'] }

The second iteration is complete and ‘C’ values are pushed into the priority queue. The new priority queue:

Priority queue	[(2 , 'C') , (5 , 'B')]

Since there are no more neighbour nodes for ‘A’, the ‘for’ loop ends and the second iteration for the ‘while’ loop begins.

The second ‘while’ loop iteration begins by **popping the first element** in the priority queue, which is ‘C’, and making it the focus node in this iteration. Following the same steps as the first iteration, the **current total distance** is compared to the **shortest known distance** in the dictionary. Then, all neighbouring nodes of ‘C’ is compared in the ‘for’ loop:

In this case, ‘D’ is compared first. Since the total distance from ‘A’ to ‘C’ to ‘D’ is less than infinity, the values in the dictionaries are updated:

Distance dict	{ 'A': 0 , 'B': 5 , 'C': 2 , 'D': 5 , 'E': inf , 'F': inf }
Path dict	{ 'A': ['A'] , 'B': ['A','B'] , 'C': ['A','C'] , 'D': ['A','C','D'] , 'E': ['A'] , 'F': ['A'] }

Same goes for the priority queue:

Priority queue	[(5 , 'B') , (5 , 'D')]

For the next iteration, the following neighbour is ‘A’ again. However, the distance from ‘A’ to ‘C’ has already been updated in the dictionary. Therefore, it is **the same** distance as in the dictionary. When compared, it is **not lesser**, which means nothing happens.

There are no other neighbours for ‘C’. Therefore, the ‘for’ loop ends.

At this point, the same sequence of steps are executed…

…

…

…

For the final iteration of the ‘while’ loop, it begins by popping the first element in the priority queue. Then, after the ‘for’ loop is finished executing and a shorter distance cannot be found, the priority queue will be empty. This means that the ‘while’ loop ends.

The starting nodes will be removed from the dictionaries because the value will be 0 anyways.

The final dictionary values:

Distance dictionary	{ 'B': 5 , 'C': 2 , 'D': 5 , 'E': 5 , 'F': 7 }
Path dictionary	{ 'B': ['A','B'] , 'C': ['A','C'] , 'D': ['A','C','D'] , 'E': ['A','C','E'] , 'F': ['A','C','E','F'] }

Then, the distance and path is returned from the dictionary:

Distance from A to F: 7

Path from A to F: A > C > E > F

With that, the Dijkstra’s algorithm is completed.

Functions in the Implementation

Besides the main Dijkstra’s algorithm function, there are three other functions for this implementation.

---

1. show_graph(list, directed, weighted)

Parameters:

list – A list of data type specifying its edges of all nodes for the graph.

directed – Boolean value to indicate whether the graph is a directed graph or undirected graph.

weighted – Boolean value to indicate whether the graph is weighted or unweighted. For unweighted graphs, all of the edges have a value of 1.

Then, a visualization of the graph is created as a Pyplot figure. If the graph is weighted, the values of the weight will be shown on the edges between the nodes. If the graph is directed, the arrow will be shown on the edges between the nodes.

---

2. show_graph_all(graph, starting_node)

Parameters:

graph – Accepts a NetworkX graph.

starting_node – Accepts one specific node in the graph.

This function utilizes the Dijkstra function to calculate the shortest distance from the starting node to each and every node in the graph. It will display a Pyplot figure, highlighting the starting node in red and showing the shortest distance and pathway from the starting node to each node in the graph.

---

3. show_graph_start_end(graph, starting_node, ending_node)

Parameters:

graph – Accepts a NetworkX graph.

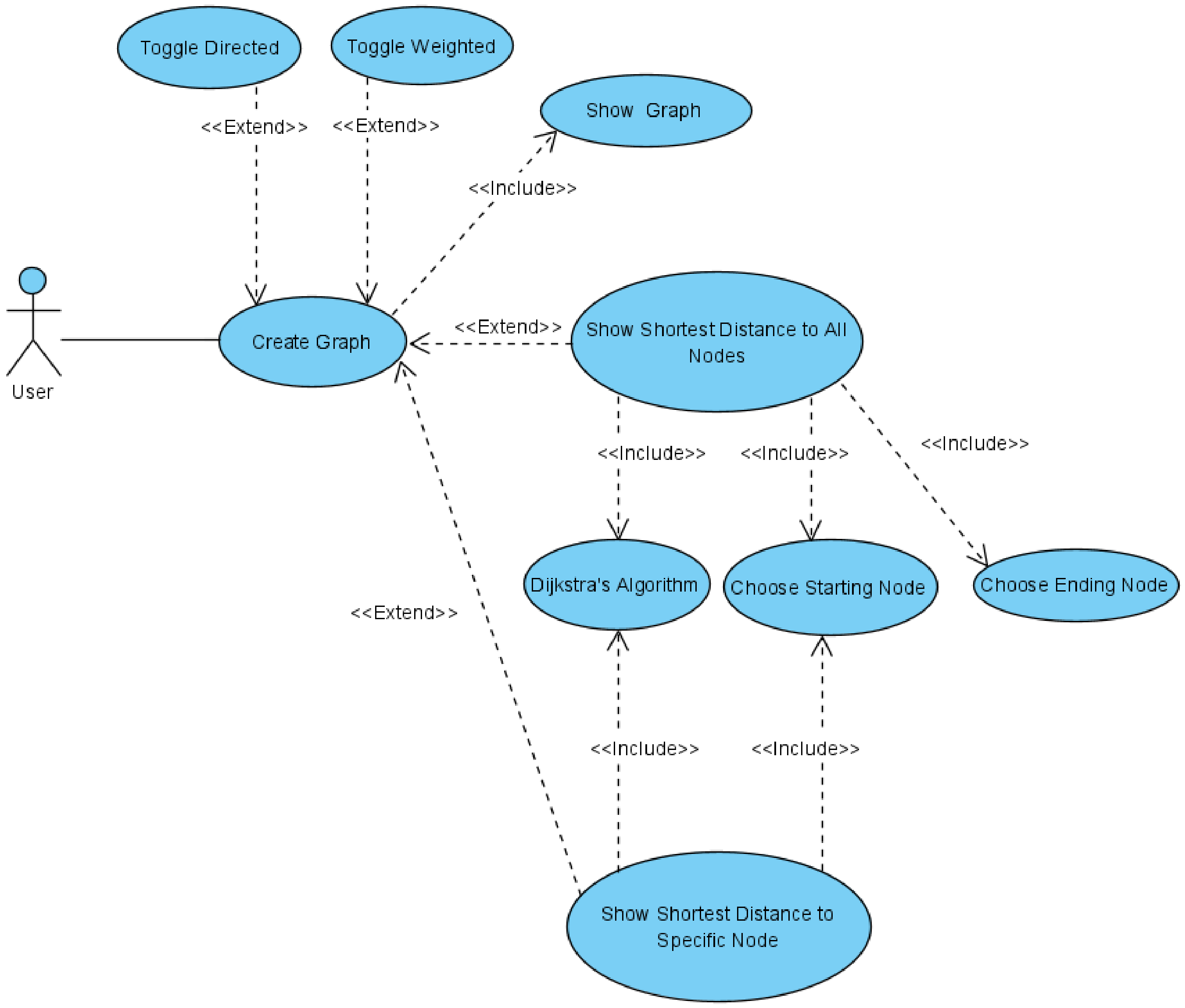
starting_node – Accepts one specific node in the graph.

ending_node – Accepts one specific node in the graph.

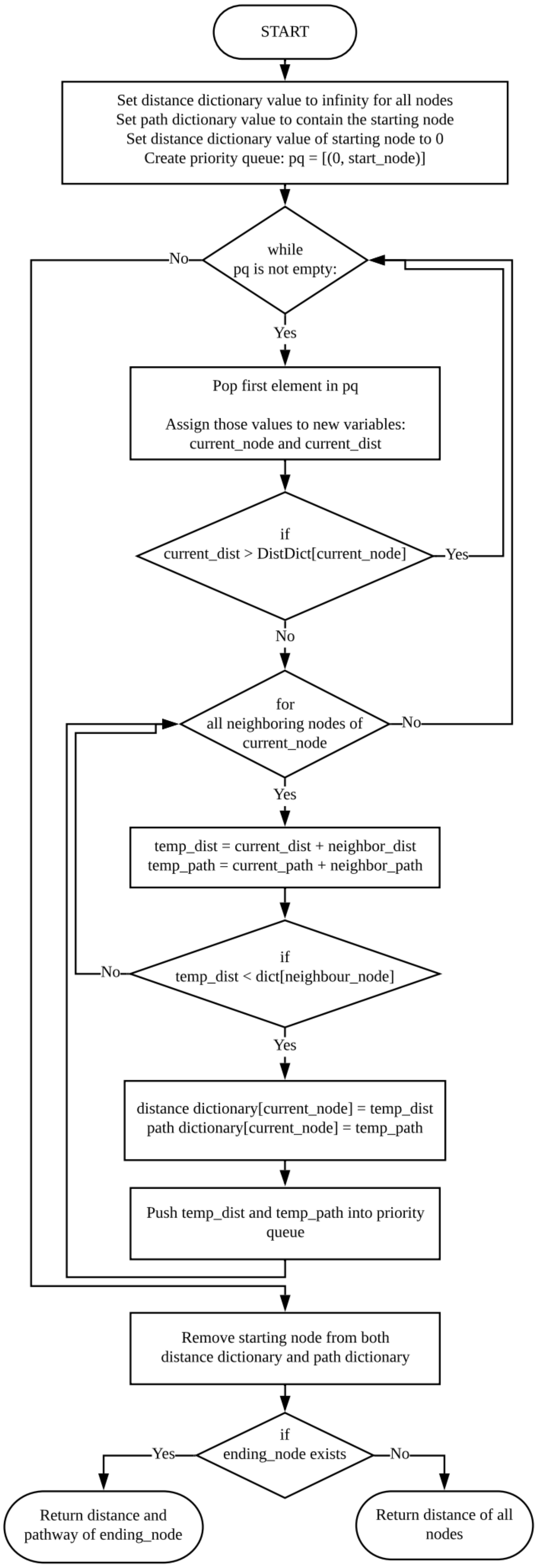
This function also utilizes the Dijkstra function to calculate the shortest path from the starting node to the ending node. The graph will be shown in a Pyplot figure with the pathway highlighted in red and the results printed at the bottom of the graph.

Diagrams for the Implementation

1. UML Use Case Diagram for Implementation of Dijkstra’s Algorithm



2. Flowchart Diagram of Dijkstra's algorithm


START
Set distance dictionary value to infinity for all nodes
Set path dictionary value to contain the starting node
Set distance dictionary value of starting node to 0
Create priority queue: pq = [(0, start_node)]
while
pq is not empty:
No
Yes
Pop first element in pq
Assign those values to new variables:
current_node and current_dist
if
current_dist > DistDict[current_node]
Yes
No
for
all neighboring nodes of
current_node
No
Yes
temp_dist = current_dist + neighbor_dist
temp_path = current_path + neighbor_path
if
temp_dist < dict[neighbour_node]
No
Yes
distance dictionary[current_node] = temp_dist
path dictionary[current_node] = temp_path
Push temp_dist and temp_path into priority queue
Remove starting node from both distance dictionary and path dictionary
if
ending_node exists
Yes
No
Return distance and pathway of ending_node
Return distance of all nodes

Additional Screenshots of the Implementation

1. Weighted and Undirected graph

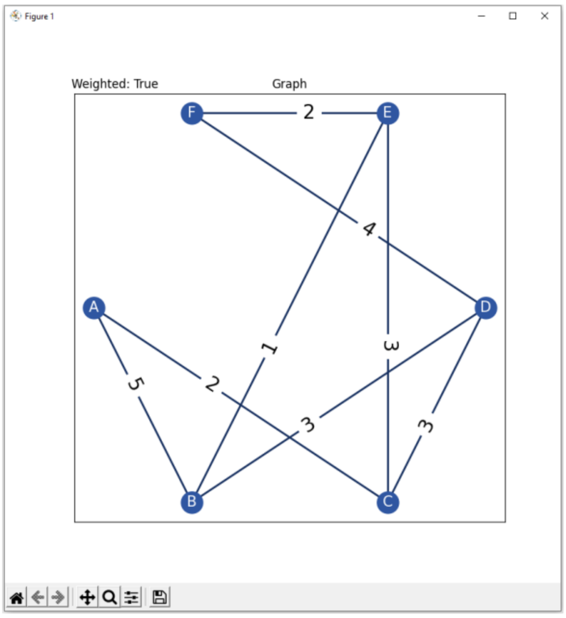
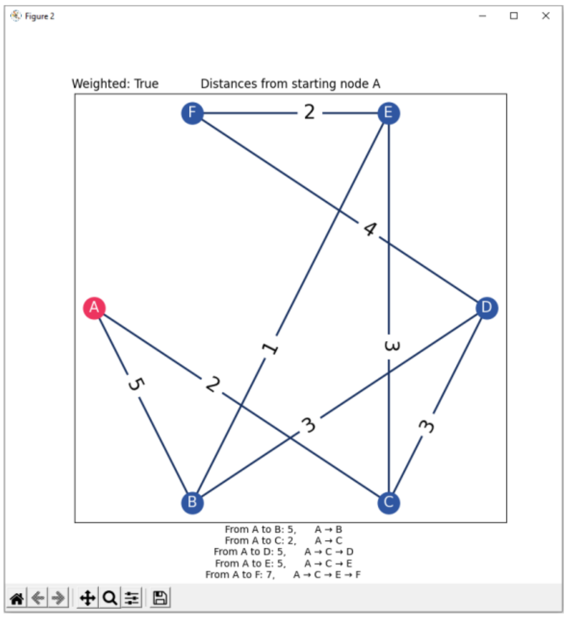
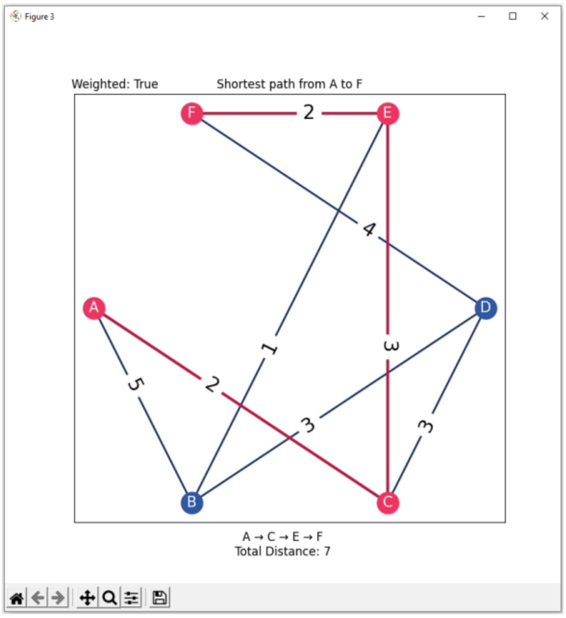


Fig 1. show_graph(G) function



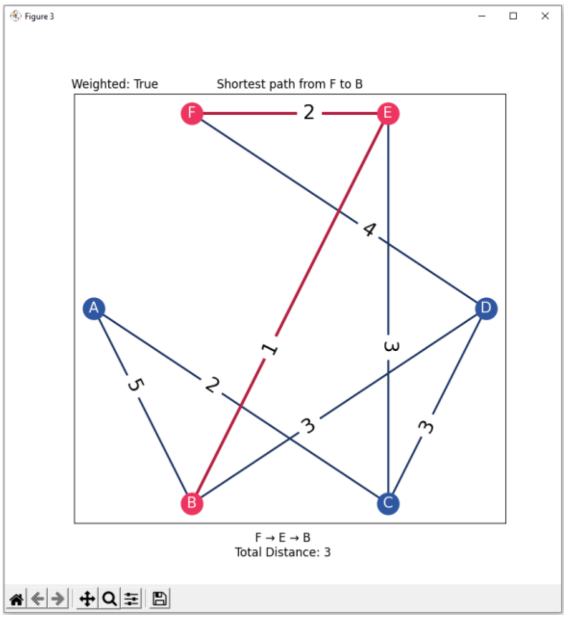
```
From A to B: 5,     A → B
From A to C: 2,     A → C
From A to D: 5,     A → C → D
From A to E: 5,     A → C → E
From A to F: 7,     A → C → E → F
```

Fig 2. Show_graph_all(G, ‘A’) function



```
A → C → E → F
Total Distance: 7
```

Fig 3. show_graph_start_goal(G, ‘A’, ‘F’) function



```
F → E → B
Total Distance: 3
```

Fig 4. show_graph_start_goal(G, ‘F’, ‘B’) function

2. Weighted and Directed graph

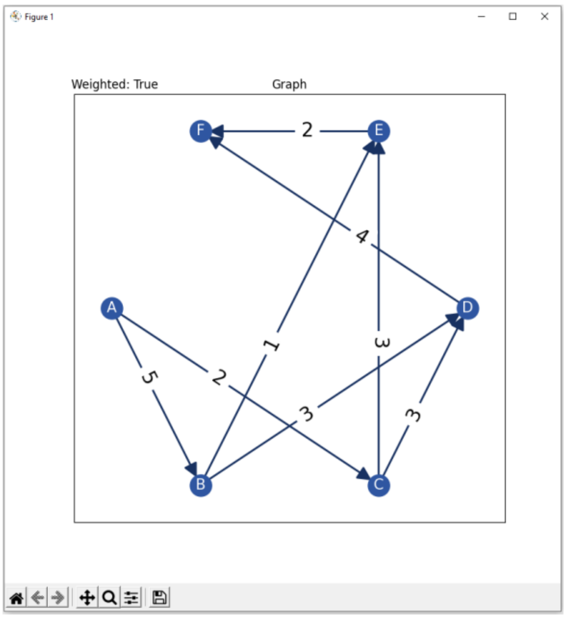
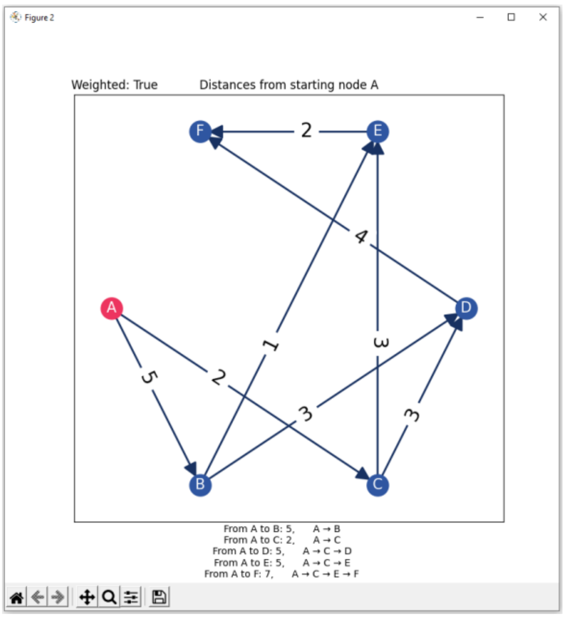
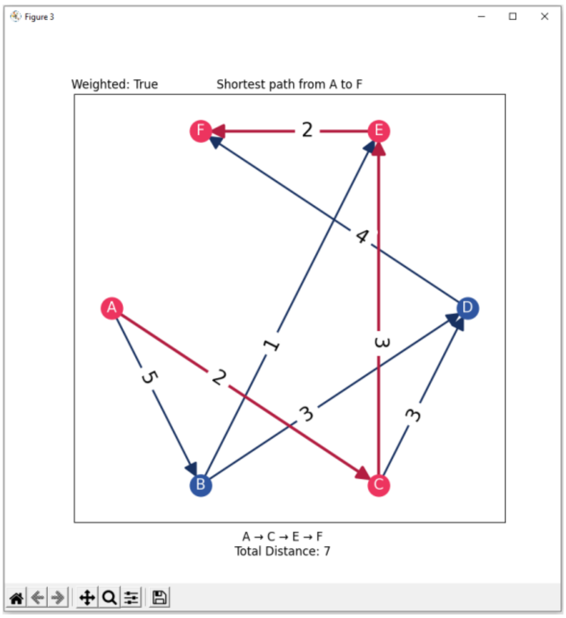


Fig 5. show_graph(G) function



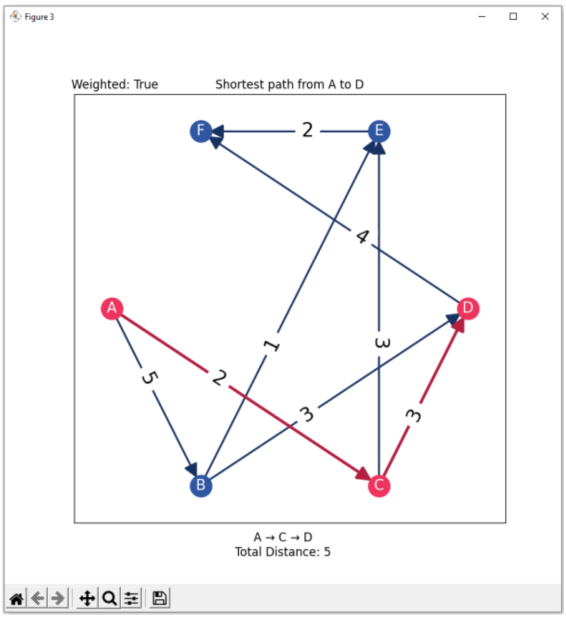
```
From A to B: 5,     A → B
From A to C: 2,     A → C
From A to D: 5,     A → C → D
From A to E: 5,     A → C → E
From A to F: 7,     A → C → E → F
```

Fig 6. Show_graph_all(G, 'A')



```
A → C → E → F
Total Distance: 7
```

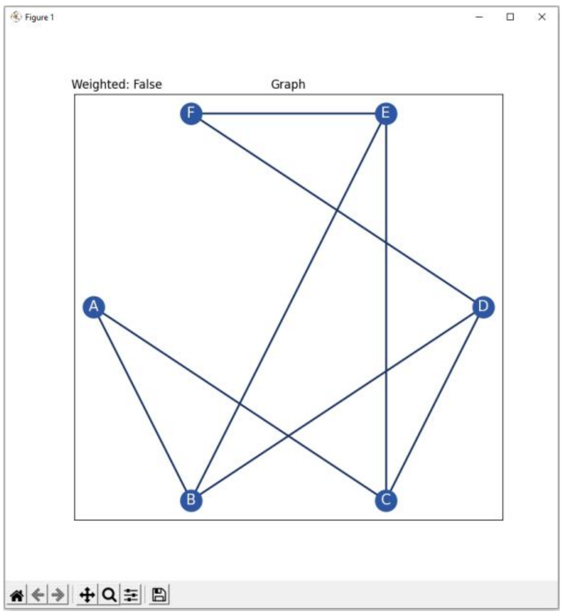
Fig 7. show_graph_start_end(G, 'A', 'F')



```
A → C → D
Total Distance: 5
```

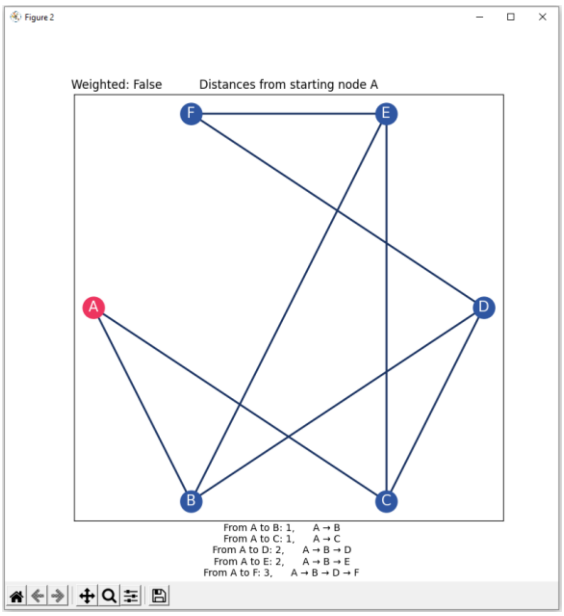
Fig 8. show_graph_start_end(G, 'A', 'D')

3. Unweighted and Undirected graph



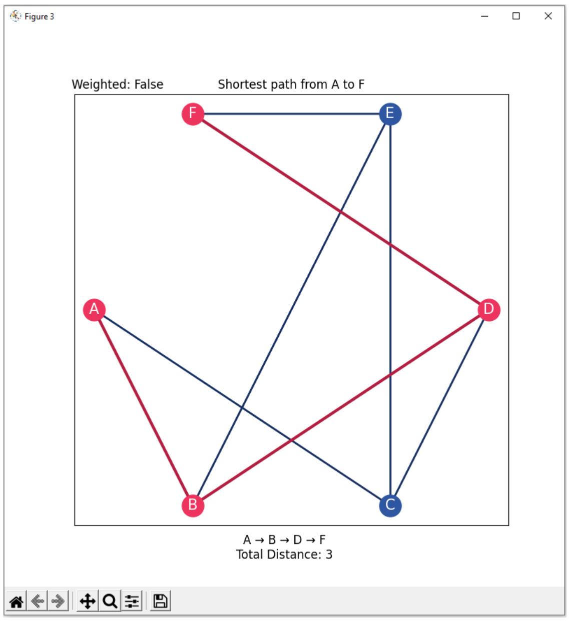
```
Weighted: False                Graph
```

Fig 9. show_graph(G)



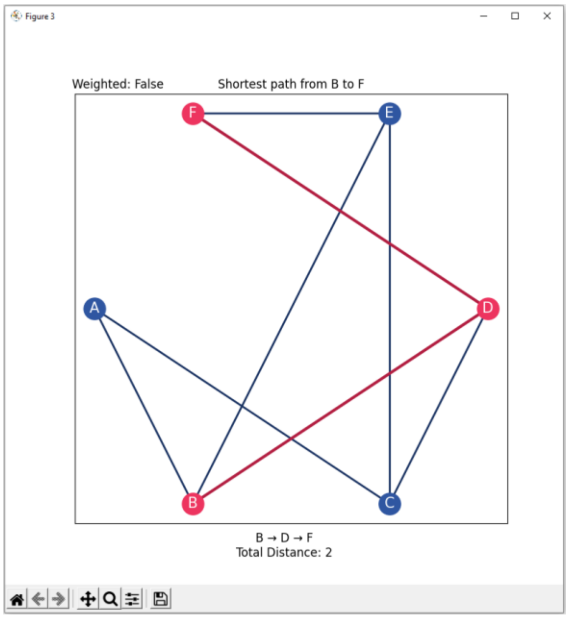
```
Weighted: False        Distances from starting node A

From A to B: 1,     A → B
From A to C: 1,     A → C
From A to D: 2,     A → B → D
From A to E: 2,     A → B → E
From A to F: 3,     A → B → D → F
```

Fig 10. show_graph_all(G, ‘A’)



```
Weighted: False        Shortest path from A to F

A → B → D → F
Total Distance: 3
```

Fig 11. show_graph_start_end(G, ‘A’, ‘F’)



```
Weighted: False        Shortest path from B to F

B → D → F
Total Distance: 2
```

Fig 12. show_graph_start_end(G, ‘B’, ‘F’)

4. Unweighted and Directed graph

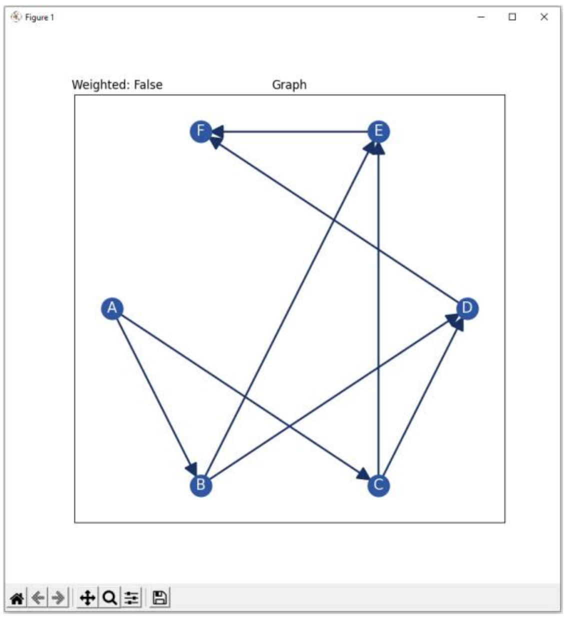
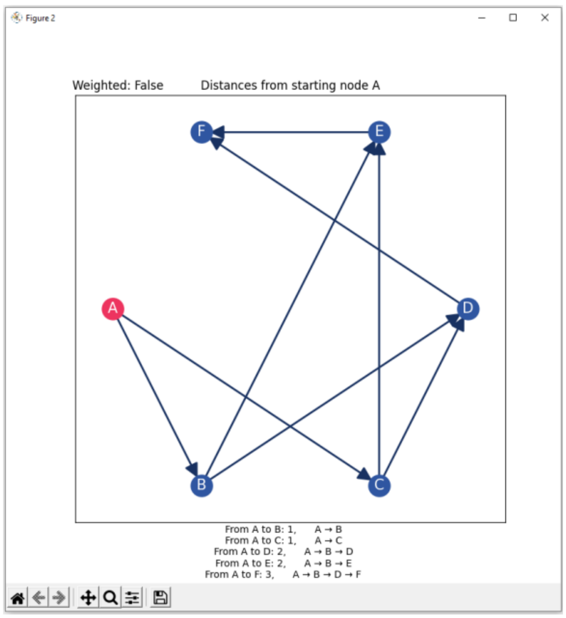
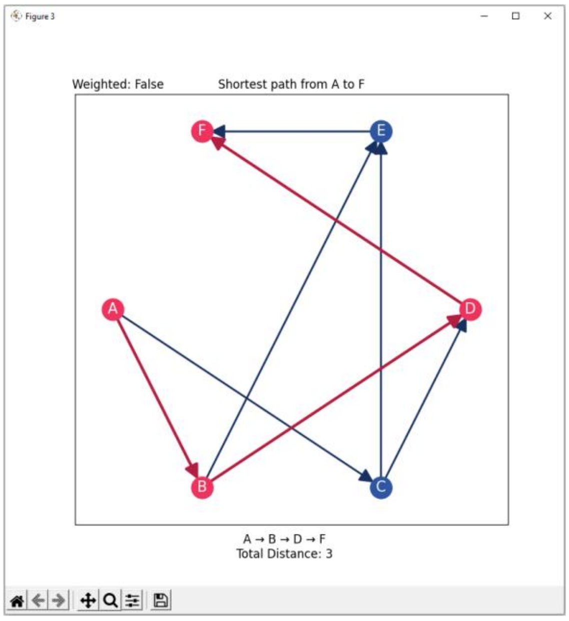


Fig 13. show_graph(G)



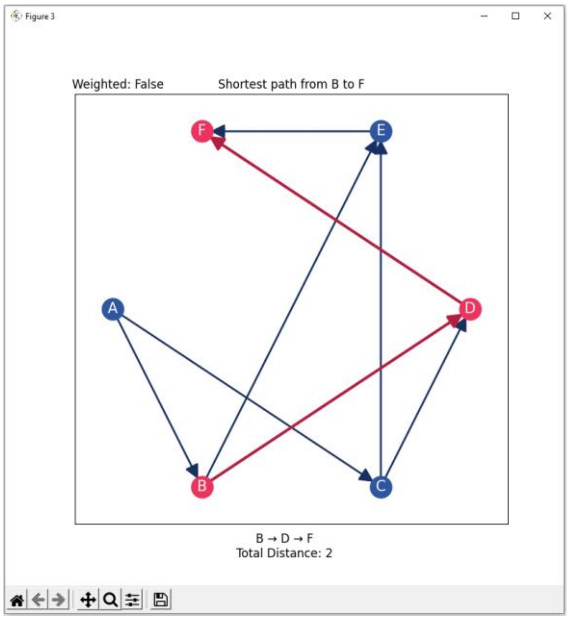
```
From A to B: 1,     A → B
From A to C: 1,     A → C
From A to D: 2,     A → B → D
From A to E: 2,     A → B → E
From A to F: 3,     A → B → D → F
```

Fig 14. show_graph_all(G, ‘A’)



```
A → B → D → F
Total Distance: 3
```

Fig 15. show_graph_start_end(G, ‘A’, ‘F’)



```
B → D → F
Total Distance: 2
```

Fig 16. show_graph_start_end(G, ‘B’, ‘F’)

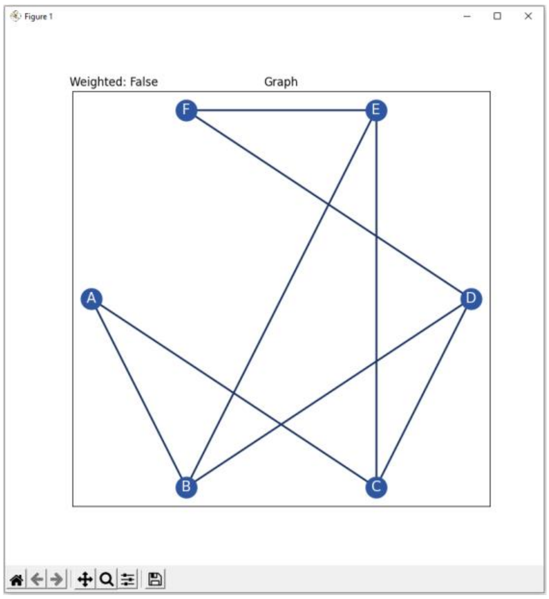
Additional Features

1. Inserting and deleting nodes and edges

After the graph is created with the edges list, it is still possible to add or remove nodes by utilizing the Python's network library. To achieve this, there are two functions.

To add a node, the `add_edge()` function can be used. The parameters are in the same order as when creating an edge in the list: `add_edge( node1, node2, weight )`

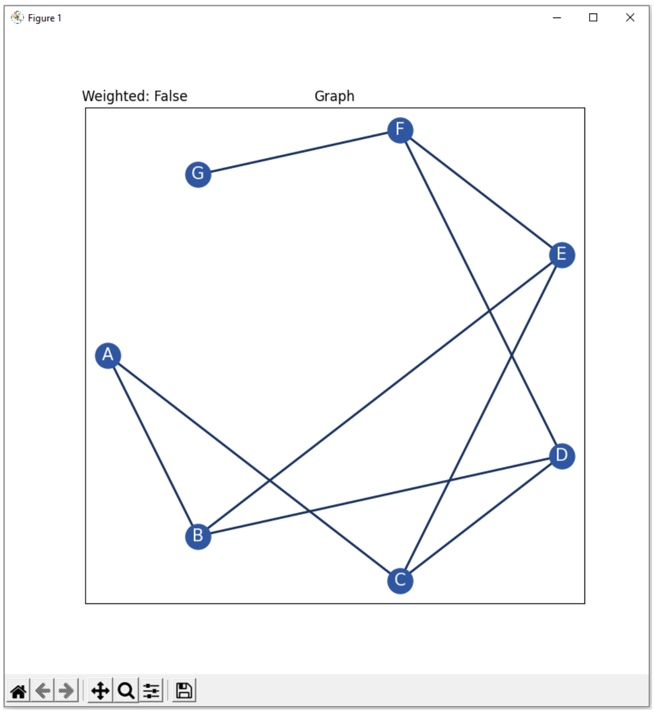
For example, a graph created with nodes A B C D E F



To add a node 'G' that links to 'F' with weight 5, the function can be used on the created graph.

```
G.add_edge('F', 'G', weight=5)
```

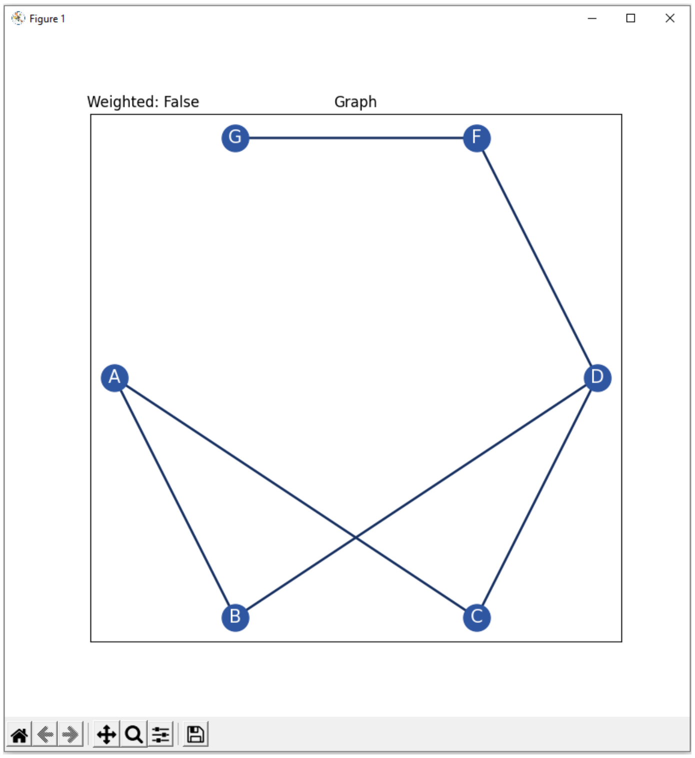
This is the updated graph:



To remove a node together with its edge, networkx only requires the node to be removed. Then, the edge will automatically be removed as well.

For example, to remove the node ‘E’ from the graph above:

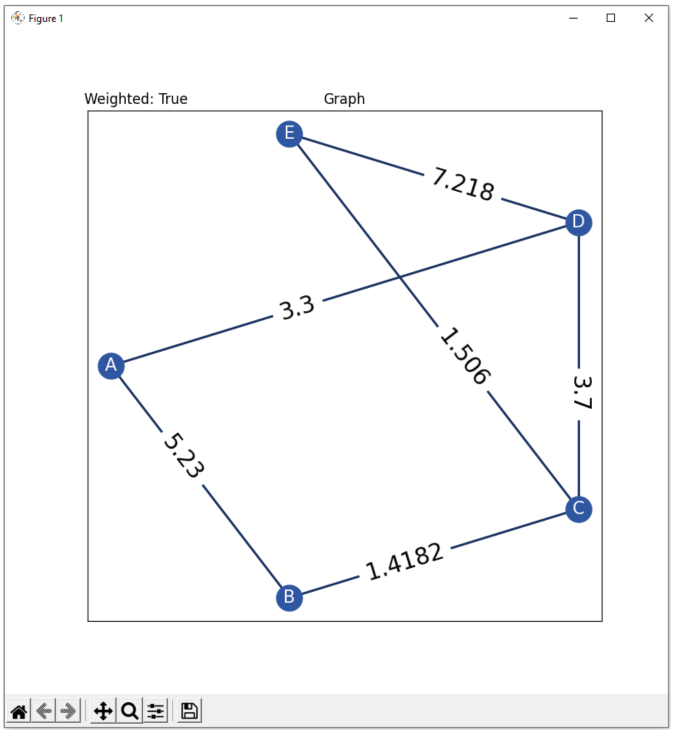
```
G.remove_node(‘E’)
```



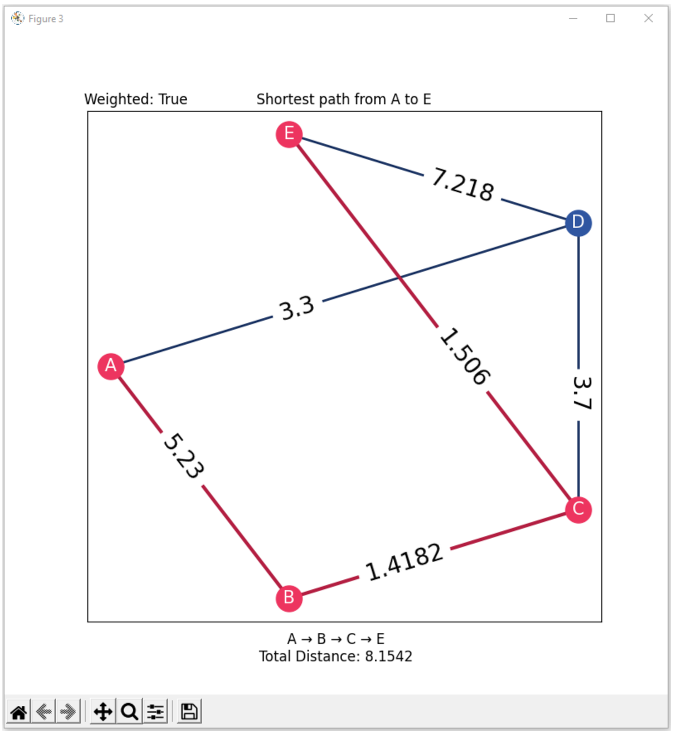
2. Support for various data types

Besides integers, the graph weight as well as its node names can support the float data type.

For example, this is a graph with weighted float data type:

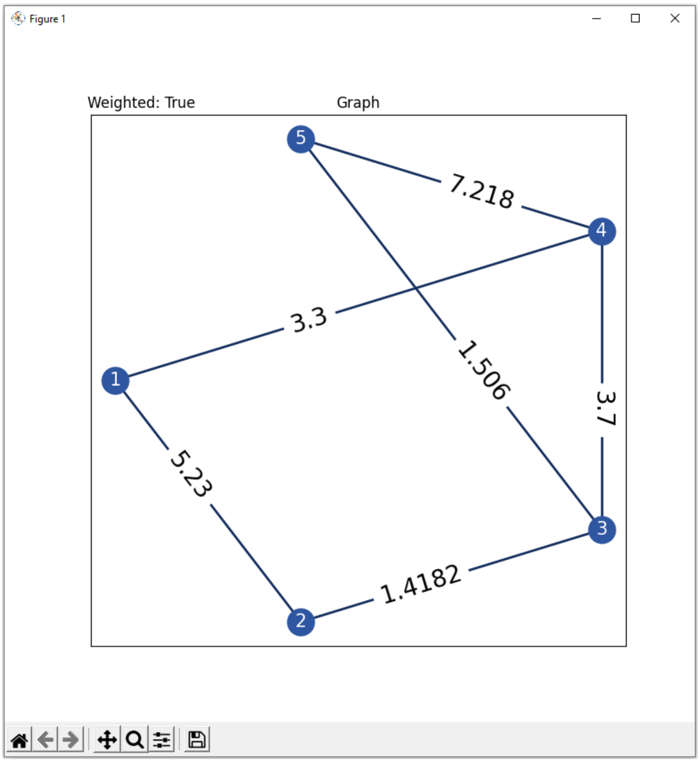


The implementation of Dijkstra's algorithm is still able to calculate the shortest path between two nodes:

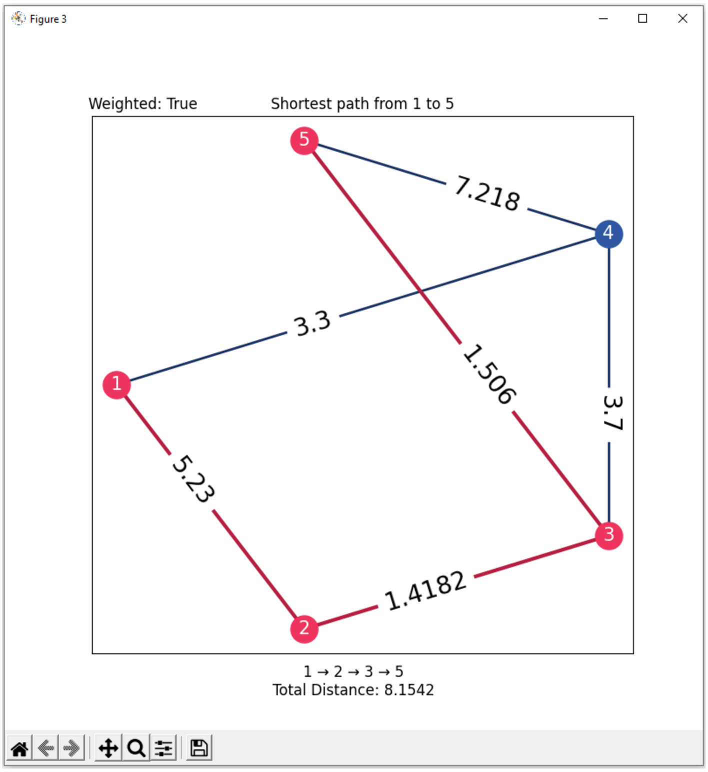


A → B → C → E
Total Distance: 8.1542

Besides that, the node names can also support integers and floats:



Finding the shortest path of an integer node graph with weighted float from node 1 to node 5:



1 → 2 → 3 → 5
Total Distance: 8.1542

Limitations of the Implementation

No negative weight

One limitation for this implementation of Dijkstra's algorithm is that it does not work when there is negative weight in the graph. With the negative weight, the implementation will be stuck and can never find the optimal pathway. To solve this problem, another algorithm must be used. For example, Bellman-Ford algorithm works well with negatively weighted graphs.

Individual Contribution and Reflection

Roger Jia Sien Boon

In this assignment, I was tasked to create binary search tree program and its documentation. I researched on the working of the binary search tree and came up the main concept of the program but later was recoded to increase its functionality and extra features that my team and I came up. After the program was done, I took the initiative to help whenever I can in the report since programming was not my strongest forte. Overall, everyone was very cooperative, and deadlines were properly met. They were also very patient and took their time to explain each of the working of the code whenever I do not understand.

Ng Wei Jinn

Throughout this assignment, my role as an all-rounder ranges from gathering teammates for meetings to developing part of the codes or focusing on the documentation. I took the initiative to gather my teammates for a voice call to discuss with them the workload distribution and schedule to ensure the assignment is completed on time with a reasonable weekly goal. During the discussions, I gave a guideline on what functionalities and features the code algorithm should include and breaking down the report requirements to understand much further on what to document, while helping with explanations of the code or any problems to my teammates in need. My responsibility was to ensure no errors were surfaced in the code algorithm and bug fix it if needed, while creating diagrams for each algorithm too. Although my full focus is to properly document or check all the explanations and requirements needed for the report. In terms of cooperation, me and my teammates were very cooperative and were able to complete all the weekly goals on time, through the help of Google Docs and WhatsApp in terms of collaboration and communicating.

Adrian Ching Liansheng

I was mainly tasked to develop the implementation for Dijkstra’s algorithm. This included the visualization of the graphs, the results, the error handling, as well as the validation of input. After doing so, I also refined it according to the suggestions of my teammates. Then, I was tasked to explain the algorithm in detail in the report. A flowchart and use case diagram were created as well to further explain the algorithm with ease. For the BST implementation, I added some additional features from the code already developed by my other teammates. Other than that, my responsibility was to compile the various sections of the report into one coherent flow, with organized content. Then, some polishing was done such as arrangement of the pictures, and table of content.

As for the collaboration efforts, I have created a rough deadline on each of the deliverables measured by weeks. This was done to increase cooperation as well as keeping track of our progress. Besides that, my teammates and I communicated during the entire process of the assignment about what tasks to achieve and delegating the tasks as well.

Challenges

While creating the program, one of our team members could not run the program even though the remaining member were able to run without problem. The message “Python: Select Workspace Interpreter” was returned after the program was run. After research, what we found is that his path variable was not setup properly after he installed python. The IDE he used is visual studio and is installed in his main drive while python was installed on the other drive. This made visual studio confused since python was detected but the path variable was not found. The solution to the problem was to change its python configuration manually and select the interpreter which was in his other drive where the python was installed instead. It was running without problem after the changes was applied.

Besides that, for the BST implementation, the validation of data types was challenging for us to accomplish. This is because our plan is to include alphabetical (A-Z), float, and int data types. When a BST is created, the data types of the following values being inserted must follow the same data type as the root node. The problem arises because in Python, user input is captured as a string. The methodology that we used to determine whether the user is trying to input a number, or an alphabet, was to use Python’s exception handling feature, “try and except”. We try converting to either int or float first. If there are no errors, this means that the user is trying to input a number. However, int and float are not the same data types. If the user wishes to input a number, it has to convert the original user input string into the correct data type (between int or float). For this, we have come up with a solution which is to detect whether the input has a decimal point. If there is a decimal point, conversion to float occurs. If there is no decimal point, conversion to int occurs.

Moreover, the visualization for both Binary Search Tree and Dijkstra algorithm was a challenge. For the BST visualization, since our algorithm was manually implemented instead of using a module and using its functions, we had to find solutions on how to create the BST visualization tree manually. Therefore, a solution that manually implements the BST visualization was found on Stack Overflow, that utilizes recursion and keep track of the size on each node and the size of children. Meanwhile, in the beginning the Dijkstra algorithm was manually coded from a reference that came with no visualization functions, but soon after that we found a python module that provided all the basic functionalities for Dijkstra which is the NetworkX Python library. Therefore, the manually coded algorithm was scrapped as the library came with functions that allowed to implement and visualize the algorithm much easily.

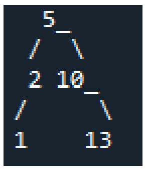
Conclusion

The implementation of BST and Dijkstra’s Algorithm was achieved through the help of the Python community, as well as its libraries. It can be further enhanced by possibly adding more features and functionalities to the current implementation. However, the most important aspect of the algorithms is the accuracy. Without an accurate implementation, the algorithms cannot serve their respective purposes. Therefore, an approach to prove the accuracy is executed.

To prove the accuracy algorithm, a comparison of results is made between the implemented code and from a website that helps to visualize the selected algorithms.

Website: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Binary Search Tree



```
   5_
  / \
  2 10_
 /     \
 1     13
```

Figure 1: Output from code

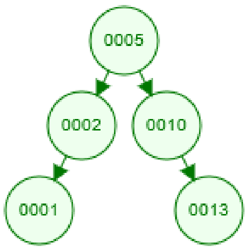
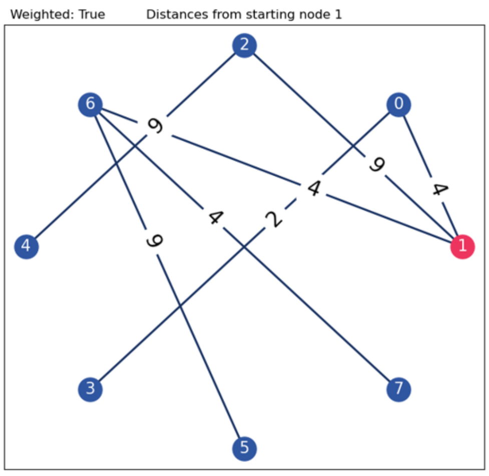


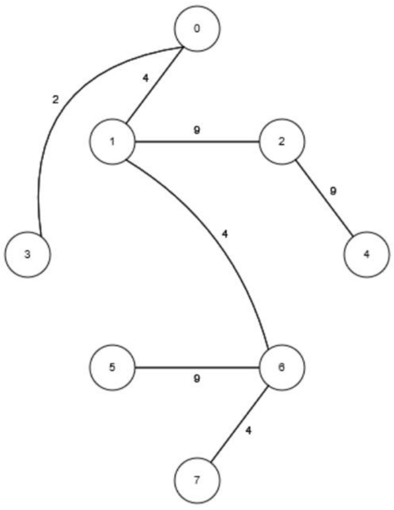
Figure 2: Output from website

Dijkstra’s Algorithm



```
 From 1 to 0 = 4,     1 → 0
 From 1 to 2 = 9,     1 → 2
 From 1 to 6 = 4,     1 → 6
From 1 to 4 = 18,    1 → 2 → 4
 From 1 to 3 = 6,    1 → 0 → 3
From 1 to 5 = 13,    1 → 6 → 5
 From 1 to 7 = 8,    1 → 6 → 7
```

Figure 3: Output from code



Vertex	Known	Cost	Path	
0	T	4	1	1 0
1	T	0	-1	1
2	T	9	1	1 2
3	T	6	0	1 0 3
4	T	18	2	1 2 4
5	T	13	6	1 6 5
6	T	4	1	1 6
7	T	8	6	1 6 7

Figure 4: Output from code

As seen from the results of comparison, we can conclude that both algorithms are implemented appropriately with the right data structures, along with all the clearly stated explanations and demonstrations in the sections above.

References

1. *Binary Search Tree - javatpoint*. www.javatpoint.com. (n.d.). https://www.javatpoint.com/binary-search-tree.
2. *Data Structure - Binary Search Tree*. Tutorialspoint. (n.d.). https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm.
3. *Dijkstra's Shortest Path Algorithm*. Brilliant Math & Science Wiki. (n.d.). https://brilliant.org/wiki/dijkstras-short-path-finder/.
4. *Drawing*¶. Drawing - NetworkX 2.5 documentation. (2020, August 22). https://networkx.org/documentation/stable/reference/drawing.html.
5. *Introduction*¶. Introduction - NetworkX 2.5 documentation. (2020, August 22). https://networkx.org/documentation/stable/reference/introduction.html#graph-creation.
6. Juan Carlos CotoJuan Carlos Coto 9. (1964, June 1). *print binary tree level by level in python*. Stack Overflow. https://stackoverflow.com/questions/34012886/print-binary-tree-level-by-level-in-python.
7. Navone, E. C. (2020, November 18). *Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction*. freeCodeCamp.org. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/.
8. Shortest Path with Dijkstra's Algorithm. (n.d.). https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/.
9. Wagner, L. (2021, June 12). *Writing a Binary Search Tree in Python with Examples*. Qvault. https://qvault.io/python/binary-search-tree-in-python/.
10. *Write a Program to Find the Maximum Depth or Height of a Tree*. GeeksforGeeks. (2021, June 3). https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/.