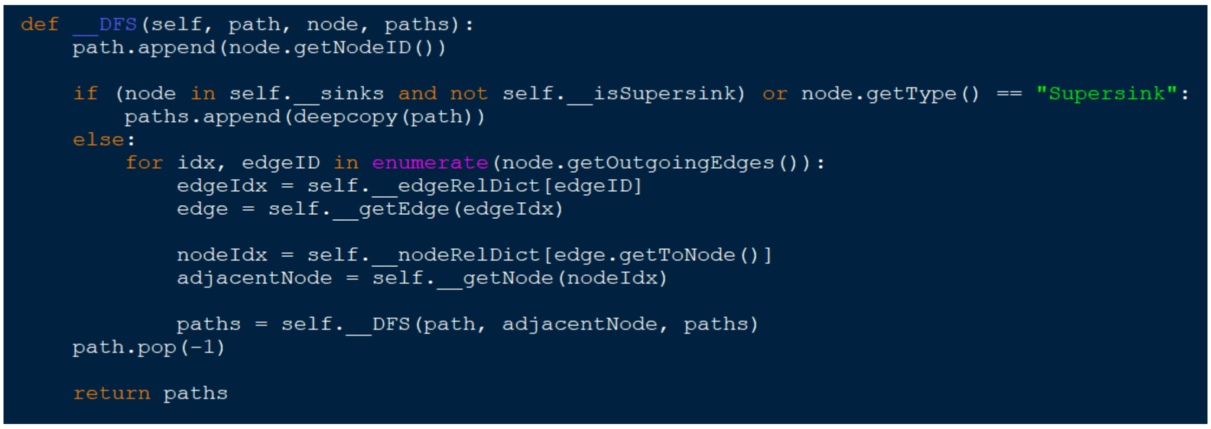
NEA Computer Science

3 Programming:

Recursive algorithms:

Recursive algorithms are used with the depth first search to get a list of all possible paths from the supersource or source, finishing at the sink or supersink, (line 748 to 764, being initially called at line 607 and 611 all within the module NetworkFlows10). This traverses the graph by using the outgoing edges within the associated arrays to get the other connected nodes, allowing all possible flows down each path to be found, both for the minimum and maximum flows.

This function, calls itself if there is an outgoing node in which hasn't been visited yet and connected to the current node, continuing along the path until no further nodes are connected, tracing backward until a node is found that hasn't been explored. Due to the path being amended only ever having the top value removed or an element pushed onto the top, the path array acts similar to a stack data structure.

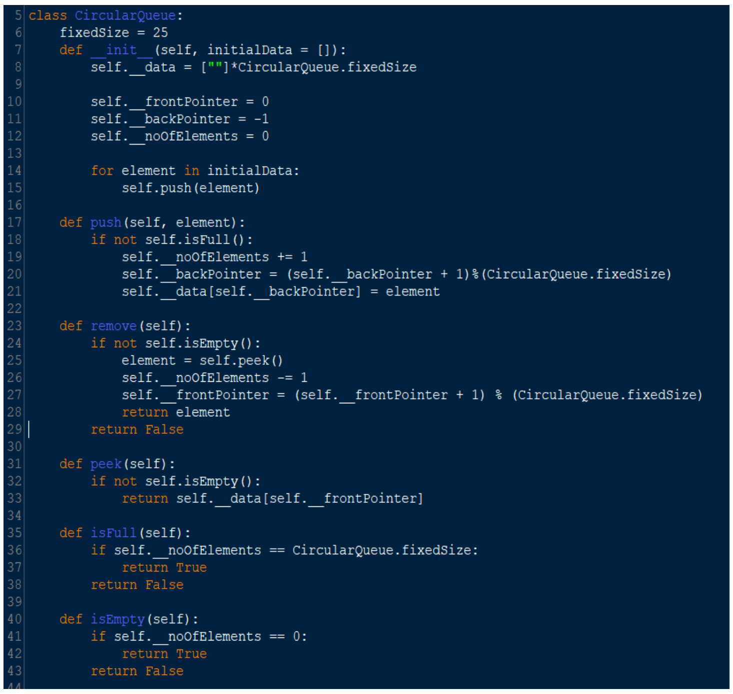


```
def __DFS(self, path, node, paths):
    path.append(node.getNodeID())

    if (node in self.__sinks and not self.__isSupersink) or node.getType() == "Supersink":
        paths.append(deepcopy(path))
    else:
        for idx, edgeID in enumerate(node.getOutgoingEdges()):
            edgeIdx = self.__edgeRelDict[edgeID]
            edge = self.__getEdge(edgeIdx)

            nodeIdx = self.__nodeRelDict[edge.getToNode()]
            adjacentNode = self.__getNode(nodeIdx)

            paths = self.__DFS(path, adjacentNode, paths)
    path.pop(-1)

    return paths
```

Queues/ Queue operations:

The Queue is implemented using a front and back pointer to identify the front and back of the data due to this being a static FIFO (first in first out) data structure. Acting as a circular queue, the elements once no more indexes are available within the list, circle back to index 0 until the list is full. This is implemented from line 5 to line 43 in the GeneralComponents module.

This is used primarily for the rotation of line colours when drawing a cut, storing the images within the array, (line 46 to 50 in the GeneralComponents module. This allows the cuts to be more identifiable by changing colours, allowing a constant colour to be available next, rotating the colour once a new cut has been created by removing an element and pushing it onto the end. This



```
 5 class CircularQueue:
 6     fixedSize = 25
 7     def __init__(self, initialData = []):
 8         self.__data = [""]*CircularQueue.fixedSize
 9 
10         self.__frontPointer = 0
11         self.__backPointer = -1
12         self.__noOfElements = 0
13 
14         for element in initialData:
15             self.push(element)
16 
17     def push(self, element):
18         if not self.isFull():
19             self.__noOfElements += 1
20             self.__backPointer = (self.__backPointer + 1)%(CircularQueue.fixedSize)
21             self.__data[self.__backPointer] = element
22 
23     def remove(self):
24         if not self.isEmpty():
25             element = self.peek()
26             self.__noOfElements -= 1
27             self.__frontPointer = (self.__frontPointer + 1) % (CircularQueue.fixedSize)
28             return element
29         return False
30 
31     def peek(self):
32         if not self.isEmpty():
33             return self.__data[self.__frontPointer]
34 
35     def isFull(self):
36         if self.__noOfElements == CircularQueue.fixedSize:
37             return True
38         return False
39 
40     def isEmpty(self):
41         if self.__noOfElements == 0:
42             return True
43         return False
```

rotation occurs in a static method changeNextColour on line 76 in the same module.

Similarly, this is also used in the breadth first search algorithm to traverse the graph when analysing a cut that has finished being drawn, (used in the function __BFS on line 2735 in NetworkFlows10). Queues are used to consider each node and add to the queue all the connected nodes that have not been visited. This removes all nodes after they have been inspected. By utilising the associated array to identify the outgoing edges of the nodes, this allows the edges and therefore the connected nodes to be found, allowing collision detection to also occur for each line segment of the cut, determining whether each node is on the source side or sink side. This also starts at the supersource and ends at a supersink if there are multiple sources or sinks and a valid position is found, else the BFS will start at the source/s and end at the sinks, accounting for the split node capacities rather than the node pre-split.

OOP inheritance:

Each node can be identified as either a source/sink or a node, depending on the outgoing and ingoing edges to that node. In order to display the difference, despite the core functionality being the same, a source/sink is drawn with a circular outline around the node shape. This is implemented through the child class SourcesOrSinks inheriting from the parent class Node, overriding the draw function, (polymorphism), which enables the visual change on the screen. The parent class Node is from line 6 to 101 and the child class SourcesOrSinks is from line 103 to 114 within the GraphElements module.

OOP aggregration:

Composition is used as a core component of the program, through the class Node, Cut and Edge which were instantiated in the NetworkFlows class. Each of the 3 classes Node, Cut and Edge, which are dynamically generated due to each graph having different numbers of each, are dependent on the NetworkFlows class to exist with the Node class objects being stored in the array __nodes and __addedNodes, the Edge class objects being stored in __edges and __addedEdges and the cut class object being stored in __cuts.

Association is used as a core component within the program, primarily with the use of buttons. Although the class Button is instantiated within the scrollbar class, to enable shifting up and down of the records, buttons are also used as stand alone component to enable features such as hiding or showing added nodes and edge such as supersources and supersinks within the NetworkFlows class. This means that the Button class is not dependent on scrollbar to exist. This is shown by the instantiation on line 103-107 in the module functionalTools3 within the scrollbar class and line 303 to 336 in the NetworkFlows class in module NetworkFlows10.

OOP dunder/accessor/mutator/static methods:

Static methods are used primarily for an ID, being auto incremented in between instantiating classes, meaning this is unique. This is seen within the class Cut and Edge, giving unique ID's. The ID within the class Cut, (line 33 to 43 within NetworkFlows10), is used to make an ID which is visible to the user to identify which cut is which, while aiding the association between the show and hide buttons. Whereas the class Edge, (line 153 to 163 in the GraphElements module), uses the ID to enable an associated relationship with the nodes in which the edge is connected to. Due to the Node class storing the outgoing and ingoing edges, rather than storing a copy of the class objects being memory inefficient, this stores the edge ID, referencing the distinct edge, which can be located. This also uses static methods to transform images without affecting the original image, allowing rescaling and re-rotating.

Similarly, classes such as Node and Cut act as interface using the accessor methods to get private attributes (line 85 to 95 in GraphElements module and 195 to 196 in the module NetworkFlows10) such as flow through a node cap or getting the outgoing or ingoing edges or even getting the cut value. Mutator methods are also used, updating the private attributes such as updating whether the cut can be drawn or update node capacity reference to identify the related index, (line 155 and 156 in the module NetworkFlows10 and line 30 and 31 in the module GraphElements). Dunder methods were used to compare a unique ID of that class object with a string, evident in both of the classes above from lines 97 to 101 in the module GraphElements and 198 to 201 in the module NewtorkFlows10.

Modules with appropriate interfaces:

Similar to how the interfaces are implemented above with accessor and mutator methods, the module functionalTools3 acts as an interface for the widgets and tools used for the user-interface, for example the scrollbars and input boxes. The scrollbars enable all data to be displayed through the use of buttons to move up and down the records, even if the data exceeds the space given, through the get element and update header functions, which are accessed from the class NetworkFlows. The input boxes also use update text functions based on events created by user inputs and get text as required. Similarly, the graph components such as the classes Node and Edge also use a modular interface, allowing access and functionality of nodes and edges, a key component of building the graph up.

Defensive programming/ exception handling:

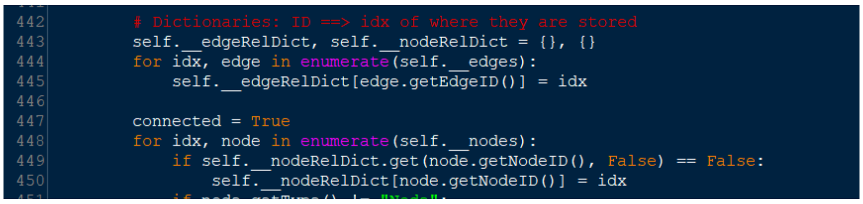
This is used primarily when entering the graph information for nodes and edges. Through the inputBox class allowing a restriction to the type of inputs, this prevents value errors when casting data type for example when an x ordinate needs to be entered, preventing any characters such as "K" being added, (line 304 to 309 and 317 to 388 in the module functionalTools3). Combined with the inputs satisfying the requirements of the object, which includes edge not going to and from the same node, this allows only valid nodes and edges to be entered, ensuring the graph can be solved, particularly with maximum and minimum capacities being entered. This is implemented in __createNode and __validateEdgeInp functions starting at lines 1482 and 1731 in the module NetworkFlows10.

Similarly, within the cuts, a Cut class can only be instantiated by a mouse click within the valid area of the graph, with further line segments being added with further mouse clicks, (line 45 to 68 within the module NetworkFlows10). By limiting this to the graph and omitting invalid cuts if source and sinks are not separated properly or inconsistencies with the side the node is on, this prevents cuts being displayed that are incorrect, allowing cut values to be added which match a valid cut, (line 2626 to 2733 in the module NetworkFlows10).

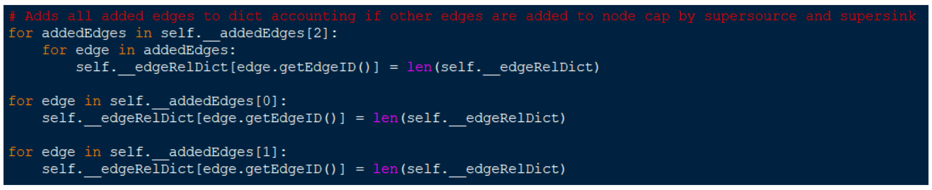
Exception handling is used particularly in the generation of the supersoures and supersinks, when finding a valid position, due to potential index errors as the list of possible positions is reduced until no more valid node position are left, (line 2438 to 2452 and 2547 to 2559 in the module NetworkFlows10).

Dictionaries/ associated arrays:

Dictionaries are used to identify any edge or node with an index that correlates to where the object is stored, enabling time efficiency when accessing these components once the graph has been made. This means a one way mapping is found between the ID's and indexes, utilised particularly when finding the edges along the paths when determining the minimum and maximum flows to be found. This is utilised in the __getNode and __getEdge functions from the lines 694 to 746 and the primary additions to the dictionary being added below, all in the module NetworkFlows10.



```
        # Dictionaries: ID ==> idx of where they are stored
        self.__edgeRelDict, self.__nodeRelDict = {}, {}
        for idx, edge in enumerate(self.__edges):
            self.__edgeRelDict[edge.getEdgeID()] = idx

        connected = True
        for idx, node in enumerate(self.__nodes):
            if self.__nodeRelDict.get(node.getNodeID(), False) == False:
                self.__nodeRelDict[node.getNodeID()] = idx
```



```
# Adds all added edges to dict accounting if other edges are added to node cap by supersource and supersink
for addedEdges in self.__addedEdges[2]:
    for edge in addedEdges:
        self.__edgeRelDict[edge.getEdgeID()] = len(self.__edgeRelDict)

for edge in self.__addedEdges[0]:
    self.__edgeRelDict[edge.getEdgeID()] = len(self.__edgeRelDict)

for edge in self.__addedEdges[1]:
    self.__edgeRelDict[edge.getEdgeID()] = len(self.__edgeRelDict)
```

User defined algorithms:

As well as the algorithms mentioned in my design for maximum and minimum flows, (line 793 to 905 in the module NetworkFlows10), finding the position of potential nodes such as supersources and supersinks, (line 2584 to 2624 in the same module), and analysing the cut, (line 2626 to 2822 in the same module), collisions are also a core function that enables the graph to remain planar. There are 6 collision detection functions, 2 within the cut class from lines 70 to 153 in the same module, to check for collisions with the line segments and the nodes and edges within the graph, allowing nodes to be determined whether they stay on the same side of the cut or flip to the other side of the cut, with the detection only specifying a collision if the line segment collides with the main arrow. The collision with the nodes is used to determine that the cut is invalid unless the node has a node capacity but has no valid position to be found. The other 4 are for determining whether a new node or edge collides with existing nodes and edges within the networkFlows class, accounting for forward and backward flow arrows of the edges, lines 1549 to 1665 and 1788 to 2011 in the same module. Through treating an edge as a 3 lines for the arrows for the main, forward and backward flow arrows as well as accounting for the label, this checks for collisions returning a boolean value.

Multi-dimensional arrays:

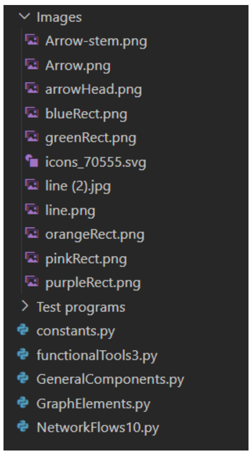
These are used for the added supersource, supersink and node capacities for both the associated nodes and edges, storing the class objects, which can be accessed in order to raise temporary minimum capacities for minimum flow paths or to get the coordinates of the centre or start and end points depending on if it was an edge or node for collision detection. This allowed, due to this being generated once the solve phase of the program has been entered, to allow resetting of these components without effecting the other nodes and arrays. Within the 2 arrays for the associated nodes and edges, index 0 was reserved for supersources, 1 for supersinks and 2 for node capacities, with multiple nodes or edges being stored under that index. This is referenced under the use of dictionaries.

Similarly, the arrays for the minimum, maximum and pathsToRemove are also multi-dimentional, storing both a path and flow per index, enabling a step by step solution to occur, (where line 690 to

692 in the module NetworkFlows10 is an example of the array being built up). These arrays therefore store the associated path and flows temporarily until they are added onto the graph, giving a priority to the pathToRemove array, followed by the minimum and then the maximum array, adding them in the order in which you would when solving the problem evident within the __addNextFlow on line 1161 in the module NetworkFlows10.

Images and file structure:

The folder for images used within the program is adjacent to the test video folder on the memory stick, which is accessed throughout the program through an os.path. The image to the right shows the file structure used, showing the image folder as a result.



```
Images
  Arrow-stem.png
  Arrow.png
  arrowHead.png
  blueRect.png
  greenRect.png
  icons_70555.svg
  line (2).jpg
  line.png
  orangeRect.png
  pinkRect.png
  purpleRect.png
> Test programs
constants.py
functionalTools3.py
GeneralComponents.py
GraphElements.py
NetworkFlows10.py
```