NEA Computer Science

2 Design:

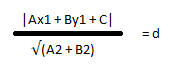
1. Explanation of the system:

The system will be developed in Python in anobject orientated format, which will be used when structuring the code. The system will use pygame, primarily taking advantage of its interactive screen, for modelling cuts, enabling the user to solve the maximum flow minimum cut theorem.

The main program will be split into 2 main components, the “Edit phase” and the “Solve phase”, dictated by the button in the top left hand corner where the graph will be drawn. The “Edit phase” primarily allows a user to build the graph up ensuring they have the all the nodes, edges and initial flows on the screen, before trying to solve the problem. Whereas the “Solve phase”, prevents any further amendments to the edges and nodes, allowing a maximum flow and minimum cut to be found.

The program will have a few core functionality tools as a part of its GUI (graphical user interface) such as buttons, scroll bars, borders and input boxes. For each of these features, each instance will follow a similar format and structure in which the tool is laid out and used. Borders will primarily identify each individual section, signifying its purpose usually through a heading. Buttons are used in a variety of ways as either stand alone components such as entering information for the nodes as well as part of the scroll bars to allow full access to all the data even if it exceeds the space given. Each button can have a single phase or dual phase meaning it will wither always signify the same function or will allow 2 different actions to occur. For example, hiding and showing a cut, supersource and supersink. Input boxes are also used with varied widths to enable user interaction, with the inputs either accepting most characters or just numbers, allowing a contrast between accepted inputs for a node ID and a X ordinate. The scroll bars can occur in 2 formats, one which structures the data into a table format and the other as one column, enabling full access to all the information within the section, whether it can all be displayed all at once or not.

Within the “Edit phase”, the user can enter values for one of three things, parameters for a node to be drawn, parameters for an edge to be drawn and parameters for a path with an initial flow along the graph to be drawn, validating each one. For the node, this takes 4 input boxes for a node ID, node capacity and the X and Y coordinates, of which will be a string, integer, float and float respectively. A node ID will be required to be unique and exclude certain reserved characters, (“T”, “S”) as well as the amended node IDs when a node initially has a node capacity. These are due to there being a possibility of a supersource, node capacity and/or supersink being created. The node capacity will be considered optional, only accepting integer keys when pressed to update the input box. The coordinates for X and Y will only accept arbitrary coordinates entered which are displayed on the graph, checking for collisions with other existing nodes or edges. A collision with a node with be found by founding the distance of the line between the 2 centres of the nodes, accounting for the radius of each node, whereas the collision with edge is found by checking if the edge goes through the centre point of the new node, else a shortest distance (d) formula is used accounting for the distance of the radius.

For the edge, this also takes 4 input boxes, the from node, to node, minimum and maximum capacities, of which will be a string, string, integer and integer respectively. For the from node and to node, a unique pair of node IDs will be required where the node ID’s are already drawn onto the graph. This also should not have an already existing edge between these 2 nodes. Each edge will have an optional minimum capacity. If this isn’t 0, (which is its default value), the minimum capacity should be less than the maximum capacity value entered, of which are both integer values. This will also check for collisions with existing nodes and edges. The collision between the node will be similar to check with a node and edge mentioned above, whereas the collision between edges will be found using the intersection point between the two edges represented as lines, validating the intersection to be within the coordinates of each edge. For each edge drawn, this will be rotated enabling the edge to go between any 2 nodes in a straight line.

For the augmented flow, this requires 2 input boxes for the path and flow, of which are a string and integer respectively. A path is required which will dictate a movement between adjacent nodes from a source to a sink, checking that the edges exist before adding the flow onto those edges. The flow must be a value greater than 0, but less than the edge which has least maximum capacity within the path.

For each node and edge entered, this will be displayed in the top left hand side of the screen in a scrollbar format in which the user can shift the displayed information up or down to view all nodes and edges using the up and down buttons. A delete button will also be corresponded for each record, allowing deletion of nodes and/or edges if values have changed, ensuring the edges connected to a deleted node is also removed.

If a node has a node capacity, once entered, the node will be split into 2, allowing the original node to be shown as well as the split node through the use of a dual phase button, which will be appear in either phase. The program therefore when solving the graph will handle either format depending on which one is displayed when finding the maximum flow and minimum cut values, accounting for the node capacity to act like limiting maximum capacity.

Within the solve phase, as the phase is initially changed, the source/s and sink/s are identified by there outgoing and ingoing edges. This is used to determine if a supersource is required, (there are multiple sources). This will check for a valid space using the collision algorithms from when nodes and edges have been entered, using generated positions for the supersource. This will then draw the supersource if a position is valid, fixing the centre point of the supersource in position, rotating the edges as required to the correct angle. If a valid position is found, this will have a corresponding button which allows the supersource to be hidden or shown to showcase the added part of the graph. The supersink, (if there are multiple sinks), will be generated in a similar way to the supersource.

By going from the source to the sink, a minimum flow and maximum flow will be found utilising if possible the augmented flows from the user which will have been entered in the “Edit phase”. Both the minimum and maximum flow will be segmented into 2 sections, a path and the flow. The minimum flow will be found by generating a list of all paths, first getting the minimum flow using the minimum capacity as the limiting factor and then finding any extra minimum flows, accounting for the flows just found, to raise minimum capacities if some edges do not have a flow equal to their minimum capacity. This creates a minimum flow which satisfies all original minimum capacities. By using the generated paths, further paths of flow will be found using the maximum capacity as the limiting factor, without using any flow assigned within the minimum flow. By adding the minimum flow and maximum flow together, the total flow through the network is found. This will be displayed in a table format within a scroll bar below the graph, with each path and flow being added to the displayed data when the next button is pressed, allowing a step by step approach, identifying each path and flow individually. This will simultaneously update the forward and backward flows to be current with the data within the scroll bar allowing the user to easier identify each new path added.

Cuts will also be enabled once the phase becomes the “Solve phase”. This allows when the mouse button is pressed within the correct area, a cut to be drawn in segments, with the last segment following the position of the mouse until the space button is clicked. Each cut drawn will have varied colour, making each cut more identifiable. This will then be analysed, checking for collisions with edges to determine both the cut value and set notation, utilising the source being the starting point, and checking if cut line has been crossed. This will use an algorithm similar to when the edge is initially added, checking collisions between existing edge and nodes. This will then display the cut and corresponding information until the “Edit phase” is re-entered including the cut notation, the cut value and which edges on the cut line goes from source to sink and sink to source. This cut information will be stored in the scrollbar in a table format similar to layout for maximum flow. An adjacent scrollbar will display the cut ID, enabling each cut to be hidden or shown when the corresponding button is pressed, simplifying the graph to focus on a specific cut if needed, following a single column format.

Once at least 1 cut has been drawn, if the cut value found is equal to the maximum flow, then a pop up border will appear in the left hand corner, stating the optimal flow has been found, ending the ability to add further cuts. This should allow the user to return back to the edit phase to amend or create a new graph to be solved.

The error and aid scrollbars will be in a single column format and be displayed in both phases of the program, displaying either key points regarding the problem itself or the key points in why an input or function has not shown such as if a node ID is not entered for any node a user has entered. An example of the aids is if there is a backward flow of 0 along an edge will be displayed on the right hand side of the window. This section will be segmented in half, allowing the user the ability to identify errors from aids.

1. HCI:

A screenshot of a computer

Description automatically generatedEach component of my HCI will follow a similar format and colour scheme, enabling easy identification of the type of functional tool it is, acting as a cornerstone for the GUI (graphical user interface). This simplistic but repetitive design will make the application more accessible to first time users or viewers, which is important given that most students will see this for the first time as it is being used. This will mean very little time will be spent understanding how the program work, allowing more time to be spent on how the program works. As well as the similar format, the tool will have similar positions in relation to other objects, enhancing the easiness to use the system as shown in the image below and explanation of each function tool below.

The borders will signify each individual section, marking as the boundaries for each individual section. The button will appear in 3 different formats. The first is a stand alone button which will perform the same action no matter the times it is clicked, evident when adding nodes, edges and augmented flow. The second will have 2 phases allowing 2 separate action to occur, usually to hide or show a specific object, while the third will be adjacent to the scrollbar shifting the records up and down. Despite there being three distinct uses for the button, each one will follow a similar rectangular shape and grey colour with solid text. The input box will be initially set as inactive, only becoming active when the user has clicked on it, setting all other input boxes as inactive. Once active, this will enable characters to be added until the input box is either inactive or the text has been reset. This although each input box will have different widths, this will follow a rectangular format with the default text being lighter than the text entered by the user. Similarly, the scrollbar will enable the shifting of records, with headers only appearing if there is data to be displayed.

The cut will also act as the final component where the user can have an interaction with the program. This allows a cut to be drawn based on the position of the user’s mouse, only changing line segments once the mouse is clicked and finishing the cut when the space bar is clicked. As this is visual when drawing, this aids the drawing as the cut is updated with the end point while drawing being at the mouse’s position. Although this isn’t accessible until the solve phase, this enables a greater variety of cuts to be made, specific to the user’s cut needed.

1. Description of modular structure of the system:

A diagram of a network flowchart

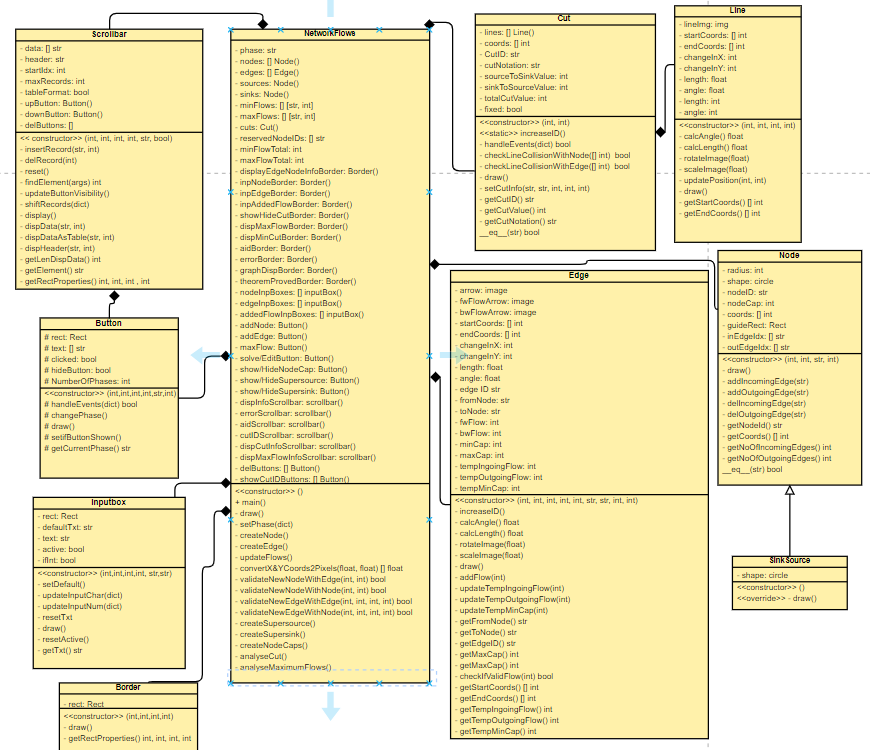
Description automatically generated

This is a hierarchical diagram displaying the key components of the solution, splitting it into 4 areas, with the inputs and graphs being the core components, enabling a graph to be built and solved.

1. Class diagram:

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

The UML shows both the functional tools for the user interface on the left and the graph components on the right, with the class NetworkFlows acting as the main class which instantiates most of the other classes.

1. IPSO:

| Class | Function name | Input | Process | Storage | Output | |
| --- | --- | --- | --- | --- | --- | --- |
|  | main |  | Sets up the loop for the screen to be displayed with a refresh rate, enabling interactivity to the screen |  |  |
| Border | \_\_init\_\_ | The coordinates of the top left hand corner of the rectangle as well as the width and height | Instantiates a rectangle of required size |  |  |
|  | draw |  | Draws rectangle object onto the screen, acting as the background for that individual section |  |  |
|  | getRectProperties |  |  |  | Returns the coordinates of the top left hand corner of the rectangle as well as the width and height as 4 integer values |
| Button | \_\_init\_\_ | The coordinates of the top left hand corner of the rectangle as well as the width and height. The text for the centre of the button as well as the number of phases | Instantiates a rectangle of required size, with the text instantiated at a centre of rectangle created with the current phase text if there are multiple phases |  |  |
|  | handleEvents | An event list which is a dictionary of all external inputs such as keys being pressed | This checks if the button is pressed by the mouse, changing the phase if the button has more than one phase |  | Returns whether the button has been pressed as a Boolean value |
|  | changePhase |  | If a button has more than one phase, this will update the current phase as well as the corresponding text that will be displayed, recentring this to the centre of the button |  |  |
|  | draw |  | This draws the centred text and the rectangle object if the button wants to be visible |  |  |
|  | setifButtonShown |  | This updates the boolean value to be the opposite of what it was, changing the button to either be hidden or shown |  |  |
|  | getCurrentPhase |  |  |  | Returns the current phase if the button has more than one phase as a string |
| scrollbar | \_\_init\_\_ | The coordinates of the top left hand corner of the rectangle as well as the width and height. A header enabling constant view of certain specified text while displayed records may vary. An optional parameter to allow text to display in a table format | Initialises the array for storing each piece of text to be displayed as well as the header, maximum records and start index for the displayed data. This will create and up and down button based on the position of the rectangle parameters given |  |  |
|  | insertRecords | The index in which to insert the element to the list of existing records and the information within that data | Inserts the element given into the array of data, allowing the start index to auto increment if the number of records is greater than the maximum records displayed |  |  |
|  | delRecord | Index within the array of data which is going to be deleted | Removes the text at that index within the array of data, which auto decrements the start index if the index is one of the indexes being displayed |  |  |
|  | reset |  | Overwrites the objects to the format in which they were in \_\_init\_\_, resetting the start index to 0 with the array of data becoming empty |  |  |
|  | findElement | The element you want to find the index of | This either returns the index, or slices the data if it is a node, checking if Node or edge ID’s match |  | Returns the index of the element you want to find from the array of data |
|  | updateButtonVisibility |  | Updates whether the up or down button is visible based on if there is data with a start index which is less than or greater than the index of data displayed |  |  |
|  | shiftRecords | An event list which is a dictionary of all external inputs such as keys being pressed | Auto increments or decrements the start index if the up or down button is pressed when they are visible |  |  |
|  | display |  | This tracks the order of data being displayed whether it is a header or a record within the array of data, so the next piece of text is a row lower within the scroll bar, resulting in a variable y position. |  |  |
|  | dispData | The element of data as well as the displayed record number of the current element | Renders the text onto the screen at the position given |  |  |
|  | dispDataAsTable | The element of data in an array format as well as the displayed record number of the current element | Renders each section of text within the array at a fixed preset distance apart in the x direction onto the screen |  |  |
|  | dispHeader | The element of data as well as the displayed record number of the current element | Renders the text onto the screen with a greater font size |  |  |
| Scrollbar continued | getLenDispData |  |  |  | Returns the length of the displayed data onto the screen as an integer |
|  | getElement | The index of an element within the array of data |  |  | Returns the element at the index within the array as a string |
|  | getRectProperties |  |  |  | Returns the coordinates of the top left hand corner of the rectangle as well as the width and height as 4 integer values |
| inputBox | \_\_init\_\_ | The coordinates of the top left hand corner of the rectangle as well as the width and height. The type of input so certain letters can be omitted as well as the default text when no text has been inputted | Initialises a rectangle of required size for text to be displayed once entered. The text initially should be set as the default text, which is centred to the centre of the rectangle.  The input box will also be set as inactive so text is not updated if keys is pressed |  |  |
|  | setDefault |  | If the input box is inactive and there is no text to be displayed from the user, updates the displayed text to be the default text |  |  |
|  | updateInputInt | An event list which is a dictionary of all external inputs such as keys being pressed | If the input box is clicked on by the mouse setting it as active, only allow keys which are integer or float related, adding the associated character into the text to be displayed |  |  |
|  | updateInputChar | An event list which is a dictionary of all external inputs such as keys being pressed | If the input box is clicked on by the mouse setting it as active, adding the associated character of the key that has been pressed into the text to be displayed |  |  |
|  | resetTxt |  | This resets the displayed text from the user to be empty, meaning default text is set to be displayed. |  |  |
|  | draw |  | This draws the input box as a rectangle with the text already entered displayed within |  |  |
|  | resetActive |  | If another input box is set to active, updates the this input box to inactive |  |  |
| inputBox continued | getTxt |  |  |  | This gets the displayed text which has been inputted by the user |
| Node | \_\_init\_\_ | The centre coordinates for the node to be displayed onto the screen, the node ID for the displayed text and the node capacity | This initialises the circular node as an object, with the centred text being the node ID, as well recording the coordinates. This also creates empty arrays for the incoming and outgoing edges and sets the added flow through node equal to 0 |  |  |
|  | draw |  | Draw a circular node without an outline on the outside, with the node ID being rendered in the centre |  |  |
|  | addIncomingEdge | The edge ID that references a specific edge | Adds the edge ID to the array of incoming edges |  |  |
|  | addOutgoingEdge | The edge ID that references a specific edge | Adds the edge ID to the array of outgoing edges |  |  |
|  | delIncomingEdge | The edge ID that references a specific edge | Deletes the element of the array incoming edges where the edge ID’s match up |  |  |
|  | delOutgoingEdge | The edge ID that references a specific edge | Deletes the element of the array outgoing edges where the edge ID’s match up |  |  |
|  | addAddedFlow | Flow being sent through node if there is a node capacity | Added the flow to the added flow if it is less than the added capacity |  |  |
|  | getNodeID |  |  |  | Returns the Node ID which is centred as a string |
|  | getCoords |  |  |  | Return the coordinates of the centre of the node as an array of 2 integers |
|  | getNoOfIncomingEdges |  |  |  | Returns the length of the incoming edges array as an integer |
|  | getNoOfOutgoingEdges |  |  |  | Returns the length of the outgoing edges array as an integer |
|  | getAddedFlow |  |  |  | Return the added flow as an integer |
| Source/sink | \_\_init\_\_ |  | Inherits the \_\_init\_\_ from Node |  |  |
| Source/sink continues | draw |  | Draw a circular node with an outline on the  outside to identify the node as either a source, sink, supersource or supersink, overriding the draw function within the class node |  |  |
| Cut | \_\_init\_\_ | The starting coordinates of the cut, (current mouse position) | This sets the cut ID and initialises the first line segment within array of all lines. |  |  |
|  | IncreaseID |  | This increases the ID of the cut for next instance of cut so each cut ID is unique |  |  |
|  | handleEvents | An event list which is a dictionary of all external inputs such as keys being pressed | This checks if the mouse button is pressed within the correct area, fixing the previous segment and adding a new instance of a line segment within the array of all lines where its start position is the last line’s end position. If the space bar is pressed this will fix the cut in place signifying it has been drawn |  | Returns whether the whole cut has been drawn or not as a boolean value |
|  | checkLineCollisionWithNode | The centre coordinates of the node | Checks if any of the line objects within the array of all lines collide with the node based on the coordinates given |  | Returns if there was a collision as a boolean value |
|  | checkLineCollisionWithEdge | The start and end coordinates of the edge | Checks if any of the line objects within the array of all lines collide with the edge between the start and end coordinates |  | Returns if there was a collision as a boolean value |
|  | draw |  | This draws the cut ID onto the screen next to the first line segment drawn |  |  |
|  | setCutInfo | The nodes on the source side of the cut as well as the nodes on the sink side of the cut, the cut values on the edges going from source to sink and sink to source as well as the total cut values | Creates objects for the cut notation, the cut value going from source to sink and sink to source as well as the total cut value |  |  |
|  | getCutID |  |  |  | Returns the cut ID as a string |
|  | getCutValue |  |  |  | Returns the total cut value as an integer |
|  | getCutNotation |  |  |  | Returns the cut notation as a string |
| Line | \_\_init\_\_ | The start coordinates and end coordinates of the line | Instantiates the image, rotating and scaling the image to the required size using the length and angle found, choosing a colour for the image |  |  |
|  | rotateImage | The angle the image needs to be rotated by | Rotates the image object by the angle found |  |  |
| Line continued | scaleImage | The length the image needs rescaling too after the line has been rotated | Rescales the image object to the length found horizontally and a fixed distance vertically |  |  |
|  | findLength |  | Use pythagoras’ theorem to get length of required image using start and end coordinates |  | Returns the length of the required image as a float |
|  | findAngle |  | Usen trigonometry to find the angle using arctan |  | Returns the angle as a float in degrees |
|  | updatePosition | The coordinates of the end position of the line | Replace the new end coordinates with the existing end coordinates, re- rotating and rescaling the image using the new length and angle found |  |  |
|  | draw |  | This draws the rotated and rescaled line object onto the screen |  |  |
|  | getStartCoordinates |  |  |  | Returns the start coordinates as an array of 2 integers |
|  | getEndCoordinates |  |  |  | Returns the end coordinates as an array of 2 integers |
| Edge | \_\_init\_\_ | The start and end coordinates of the edge as well as the 2 node ID’s the edge is linking together as well as the minimum and maximum capacity | Rescales and rotates the images for forward flow, backward flow and main arrows, instantiating each image along with the 2 node ID’s, the minimum and maximum capacities. The forward and backward flow is also determined, while added flow values are set to 0 |  |  |
|  | increaseID |  | Increases the ID so that all edges are unique when instantiated |  |  |
|  | findLength | The length the image needs rescaling too after the line has been rotated | Rescales the image object to the length found horizontally and a fixed distance vertically |  |  |
|  | findAngle | The angle the image needs to be rotated by | Rotates the image object by the angle found |  |  |
|  | rotateImage | The angle the image needs to be rotated by | Rotates the image object by the angle found |  |  |
|  | scaleImage | The length the image needs rescaling too after the line has been rotated | Rescales the image object to the length found horizontally and a fixed distance vertically |  |  |
| Edge continued | draw |  | This draws the rotated images of the main, forward and backward arrows, with the corresponding text of either the minimum and maximum capacities, the forward flow or the backward flow, centring this at the middle of the arrow |  |  |
|  | addFlow | The flow which is going to be added | The flow is added to the forward flow and removed from the backward flow |  |  |
|  | updateAddedFlow | The flow you have assigned to that edge | Update the added flow by the value given |  |  |
|  | resetAddedFlow |  | Set added flow to 0 |  |  |
|  | updateTempMinCap | The updated minimum capacity value | Updates the temporary minimum capacity value. |  |  |
|  | getEdgeID |  |  |  | Return the edge ID object as a string |
|  | getFromNode |  |  |  | Returns the node in which the edge is coming from as a string |
|  | getToNode |  |  |  | Returns the node in which the edge is going to |
|  | getMaxCap |  |  |  | Returns the maximum capacity object as an integer |
|  | getMinCap |  |  |  | Returns the minimum capacity object as an integer |
|  | checkIfValidFlow | Flow that is trying to be added | Checks if the forward flow + the additional flow is valid within that edge |  | Returns whether the additional flow is possible as a boolean |
|  | getStartCoordinates |  |  |  | Returns the start coordinates as an array of 2 integers |
|  | getEndCoordinates |  |  |  | Returns the end coordinates as an array of 2 integers |
|  | getAddedFlow |  |  |  | Returns the added flow as an integer |
|  | getTempMinCap |  |  |  | Returns the temporary minimum capacity as an integer |
| Network flows  Network flows continued | \_\_init\_\_ |  | This instantiates all the borders for each section of the program as well as the input boxes for the nodes, edges and augmented flows to be entered, setting the phase as the “Edit phase”. The buttons and scrollbars are also all instantiated along with 7 empty arrays for the edges, nodes, sources, sinks, cuts, maximum flows and minimum flows |  |  |
|  | main |  | This calls most of the functions within the program depending on the phase set, due to some objects only being displayed in a particular phase such as cuts being drawn in the ”Solve phase” or all the time such as the borders for each section. However, it will analyse the buttons present if they are in relation to a record of a scrollbar |  |  |
|  | draw |  | Objects are drawn depending on the phase set, with each draw function for each object being called or text being rendered onto the screen. For example, the text includes the heading for the input boxes for information regarding the nodes, edges and augmented flow |  |  |
|  | setPhase | An event list which is a dictionary of all external inputs such as keys being pressed | This checks if the solve/edit button has been pressed, changing allowing the current phase of the button to be changed |  |  |
|  | createNode |  | This gets the node ID, node capacity and arbitrary coordinates from the input boxes, adding a node once the inputs have been validated into the array of nodes. These checks may include that the arbitrary coordinates entered are within the given range as well as that the new node does not collide with any existing nodes or edges |  |  |
| Network flows continued | createEdge |  | This gets the node ID’s for where the edge is going from and to as well as the minimum and maximum capacities from the input boxes, adding the edge if there are no errors into the array of edges. This will update the incoming and outgoing edges for each node that the edge is connected to. These checks may include that each node ID exists and that the minimum capacity is greater than or equal to the maximum capacity. The edge should also be checked to see if there are any collisions with existing nodes or edges. |  |  |
|  | updateFlows |  | This gets the path and flow from the input boxes and validates the inputs, adding the flow to each edge if no errors found. These checks may include that all edges exist and that the flow added does not exceed a maximum capacity of any edge |  |  |
|  | convertX&YCoords2Pixels | The X and Y coordinate as arbitrary floats | This converts the arbitrary position of the coordinates to the associated X and Y positions on the screen |  | Return the updated coordinates as a tuple containing 2 floats |
|  | validateNewNodeWithEdge | The centre coordinates for the node | Checks if any of the edge objects within the array of all edges collide with the node based on the coordinates given |  | Returns whether there was a collision as a boolean value |
|  | validateNewNodeWithNode | The centre coordinates for the node | Checks if any of the node objects within the array of all nodes are within a fixed distance the new node based on the coordinates given |  | Returns whether the node was too close to another node as a boolean value |
|  | validateNewEdgeWithEdge | The start and end coordinates of the new edge | By treating the edges as lines, this checks if any of the edge objects within the array of all edges collide with the new edge within the coordinates of the start and end points given |  | Returns whether there was a collision as a boolean value |
|  | validateNewEdgeWithNode | The start and end coordinates of the new edge | Checks if any of the node objects within the array of all nodes collide with the new edge between the coordinates given |  | Returns whether there was a collision as a boolean value |
| Network flows continued | createSupersource |  | By finding a valid position, this instantiates a node and the edges needed to connect the supersource to the sources, adding these to the array of nodes, sinks and edges. This should add the incoming and outgoing edges to each node that the added edges are connected to. Checks for valid position including checking that the new node and edges do not collide with existing nodes and edges |  |  |
|  | createSupersink |  | By finding a valid position, this instantiates a node and the edges needed to connect the sinks to the supersink, adding these to the array of nodes, sinks and edges. This should add the incoming and outgoing edges to each node that the added edges are connected to. Checks for valid position including checking that the new node and edges do not collide with existing nodes and edges |  |  |
|  | createNodeCaps |  | If a node has a node capacity, the 2 new nodes will be instantiated with an edge acting as the limit to the flow once a valid position is found, linking this to the node within the array of nodes. Checks for valid position include checking that the new nodes and edges do not collide with existing nodes and edges |  |  |
|  | analyseCut |  | If the cut has been fixed, by traversing the graph from source to sink, using the collision with the line segments to identify which side of the cut a node is on, a cut notation will be found as well as the cut value from source to sink and sink to source as well as the total cut value. Checks for a valid cut including the line segments of the cut splitting the graph in 2 |  |  |
|  | analyseMaximumFlows |  | Minimum flows will be found, with the minimum capacity first acting as a limiting factor before attempting to raise raising the temporary minimum capacity of edges if needed. The maximum flow will then be found by trying to add further flow into the graph, generating a value for the total flow through the graph. |  |  |

This IPSO aligns with UML above, matching the classes and structure of the program.

1. Identification of Validation required:

|  |  |  |
| --- | --- | --- |
| Data item | Check Type | Check Details |
| Node ID | Length | Must be able to fit within the input box and within the node when drawn |
| Node ID | Duplication of Node | Must be able to uniquely identify each node |
| Node Capacity | If positive integer | Checks if positive integer for limit of flow sent through the node |
| Arbitrary x position | If float | Checks to see if this a positive float so the node can be drawn on the screen |
| Arbitrary x position | Given range | Checks to see if the integer is within the coordinate grid displayed on screen to ensure it is visual |
| Arbitrary y position | If float | Checks to see if this a positive float so the node can be drawn on the screen |
| Arbitrary y position | Given range | Checks to see if the integer is within the coordinate grid displayed on screen to ensure it is visual |
| Area covered by new node position | Collision detection | Checks if the node position is within a certain distance of an existing node to enable visible edges if added y user n later inputs |
| Area covered by new node position | Collision detection | Checks if the node position collides with an existing edge to maintain the graph as planar |
| Edge – From which Node | Valid node identifier | Checks if node identifier given exists |
| Edge – To which Node | Valid node identifier | Checks if node identifier given exists |
| Edge – minimum capacity | If positive integer | Checks if positive integer for lower bound for flow on the edge |
| Edge – maximum capacity | If positive integer | Checks if positive integer for upper bound for flow on the edge |
| Edge – minimum flow and maximum flow | Comparison between minimum flow in/out and maximum flow out/in | Checks if a flow can be found, (minimum flow must be less than maximum flow) |
| Area covered by an edge | Collision detection | Checks if a new edge collides with a existing edge so the graph remain planar |
| Area covered by an edge | Collision detection | Checks if a new edge collides with an existing node so the graph remains planar |
| Augmented flow - path | Valid Edge | Checks if all edges within the path exist |
| Augmented Flow - flow | If positive integer | Checks if the flow entered can be added to forward flow for each edge |
| Augmented flow - flow | Valid flow | This checks that the flow entered + the forward flow does not exceed the maximum capacity for each edge within the path. |
| Solve Button pressed | If pressed | Checks if mouse is pressed within the given area, checking a valid graph is drawn with a source and sink |
| Button for maximum flow in steps | If pressed | Checks if graph is drawn and there is another path and valid flow greater than 0 to be added |
| Minimum cut - drawn | Position | Checks if the cut wanting to be drawn is within the area for drawing the graph |
| Minimum cut - drawn | Valid Cut - Position | Checks if the cut is valid – source/s and sink/s are opposite sides and cut does not end within the graph |

The validation above is primarily due to the user entering the inputs for nodes and edges, making up all parts of the problem. This therefore enables the problem to be executable, allowing the ability to catch errors such as value error or even in terms of the problem for the cuts.

1. Algorithms:

7.1) Drawing the graph:

Get a supersource/supersink: This algorithm finds the position of either a supersource or supersink based on there being multiple sources/sinks for this to be displayed.

1. For all sources or sinks, find the average of the x ordinate and y ordinates, taking this as the starting possible position of the supersource/ supersink.
2. Set change to be an array containing the ability for the start position to shift along the x and y axes in either the positive or negative direction.
3. Set the counter = 2 due to the div by 2, to mark the number of times changes are to be made to the coordinate, signifying different positions.
4. While a position is not found, move the potential position in a spiral pattern, trying each possible position to see if there are collisions:

* If the counter MOD 2 = 1, then update the array change, removing the first element and appending to the end.
* Set the new changes in x and y through the new first element of change.
* Add the changes to temporary position of the node, checking to see if there are no collisions and that position is within the area where nodes and edges can be drawn. If there are no collisions, then the position is found. Repeat this step the counter DIV 2 number of times.
* Add 1 to counter.

A white background with black text

Description automatically generated

7.2) Maximum Flow:

Maximum Flow: This algorithm updates the minimum capacity of each edge, to create the minimum paths required and additional paths subject to the restraints of the minimum capacity of each edge and the maximum capacity, splitting the overall flow into two segments.

1. Get a list of all paths via a depth first search:

* Starting at the source/supersource and the path being an empty array, call the subroutine, appending the parameter node to path.
* Get adjacent nodes which have not been visited via the outgoing edge and call the subroutine, creating recursion.
* Check if the node is the sink/supersink, if so, append the path to paths.
* A screenshot of a computer code

  Description automatically generatedRemove final element to enable multiple paths to be created.

1. Get the minimum flows from each path:

* Iterate through each edge within a path, if the temporary minimum flow has not been assigned or if the minimum flow of the edge – added flow from previous paths < temporary minimum flow, set the temporary minimum flow to the minimum flow of that edge – added flow from previous assigned paths.
* If the temporary minimum flow > flow through the node capacity, then assign temporary minimum flow equal to flow through the node capacity.
* Update for each edge the forward flow, adding on the temporary minimum flow as well as the backward flow.
* If the temporary minimum flow is greater than 0, append to minimum paths, allowing the ability to track paths. With the flow in this path being added to each edge and node capacity as marker for future paths.

1. Get the minimum paths and flows where the minimum capacities are raised:

* While there are paths being found that can have the temporary minimum capacities raised, set the best possible path and flow to [] and iterate through all the paths:

1. For each edge in the path, if the maximum flow left has not been assigned, set the maximum flow left to be the maximum capacity of the edge – the flow assigned to the edge already and set the temporary minimum flow to be the minimum capacity of the edge – flow assigned to the edge.
2. If the minimum flow available for that edge is > than the temporary minimum flow and the minimum flow available for that edge – the temporary minimum flow <= maximum flow left, then the temporary minimum flow is set to the minimum flow of the edge and the maximum flow left is reduced by the difference the temporary minimum flow has increased by.
3. If the minimum flow available for that edge is > than the temporary minimum flow and the minimum flow available for that edge – the temporary minimum flow > maximum flow left, then the temporary minimum flow is increased by the maximum flow left and the maximum flow left is reduced to 0.
4. If the maximum capacity of the edge – flow assigned to the edge < maximum flow left, then reduce the maximum flow left to the maximum capacity of the edge – the flow assigned to the edge.
5. A white background with text

   Description automatically generatedIf the maximum capacity of the edge – the flow already sent down the edge – the temporary minimum flow < 0, then the temporary minimum flow is decreased by the difference and the maximum flow left is increased by the difference between the minimum flow possible through the edge – the temporary minimum flow – maximum flow left.
6. For each node in the path, if the node capacity value – the already added flow through the node capacity is < than the temporary minimum flow, temporary minimum flow is the node capacity value – the already added flow through the node capacity, with the maximum flow left being sent to 0.
7. If the path found with a temporary minimum flow greater than 0 has more edges with possible minimum flow to increase or a best possible path and flow have not been found yet, then set the best possible flow and path as the flow and path just found.

* The added flow in this path is added to each edge as a marker for future paths. If there is a best possible flow and path, assign this to the minimum paths and repeat until no more paths and flow are added to minimum paths.

1. Get the maximum flows from each path:

* Iterate through each edge within a path, if the temporary maximum flow has not been assigned of if the maximum flow of the edge – added flow from previous paths < temporary maximum flow, set the temporary maximum flow to the maximum flow of that edge – added flow from previous assigned paths.
* If the temporary maximum flow > flow through the node capacity, then assign temporary maximum flow equal to flow through the node capacity.
* Update for each edge the forward flow, adding on the temporary maximum flow as well as the backward flow.
* If the temporary maximum flow is greater than 0, append to maximum paths, allowing the ability to track paths, with the flow in this path being added to each edge as marker for future paths.

1. If there is a path that has a pre-augmented flow:

* Set the removeInitialFlow to be an empty dictionary.
* If the path is in the minimum and maximum paths and if the flow of the augmented path is greater than the flow within the minimum and maximum paths.
* Find the difference between the two flow values, and append the path and flow to removeInitialFlow, with the flow acting as the negative difference.
* If the flow of the minimum or maximum path becomes 0, remove the path from minimum or maximum paths.
* If the flow of the minimum or maximum paths is < than the augmented flow entered, a new path is created with a negative flow to subtract the difference from the path so it matches the maximum and minimum flows found.

7.3) Minimum cut:

Drawing the cut: This algorithm allows the user to draw a single cut onto the graph at a time, ensuring that cut drawn is valid both within the correct area on the screen and in respect to the graph itself.

1. Get the mouse position and check if this is within the area where the graph can be drawn onto the screen, comparing through the use of coordinates.
2. Check if mouse is clicked to create a cut, beginning with a segment. If a cut is already created, a segment will be made continuing from the end position of the previous segment using this as the start position. With regards to the previous line segment, this has its parameters fixed.

* Set initial position of the segment to the coordinate position of the mouse when clicked. This will act as fixed initial point of the segment.
* Set end position of the segment to the current position of the mouse. This is continuously updated until step 4 or 5.

1. A triangle with text below

   Description automatically generated with medium confidenceDraw segment onto the screen by finding an angle and length based off the current coordinates, updating this every refresh of the screen. This allows the segment to be viewed by the user by transforming the length and angle of the segment to the horizontal.

* For length, find change in x and y in regard to start and end positions of the segment. To find the shortest distance, the Pythagoras is used where the change in y and x act as the vertical an horizontal sides perpendicular to each other.
* For the angle, due to the change in x and y getting the horizontal and vertical sides, SohCahToa can be used, specifically tan-1(change in y / change in x). If this is undefined, the line is vertical, meaning based on whether the change in y is positive or negative, will dictate direction. For example, as pygame takes its starting coord from the top left hand corner, a positive change in y means the segment is going directly downwards.

1. If space button is pressed, delete the current segment fixing the cut in place, validating the last fixed position, (where the mouse has last been clicked) to be within the area.

Analysing the cut: This algorithm analyses the cut and ensures that it is valid, meaning that the source/s and sink/s are on opposite sides. By getting the nodes going from source to sink and sink to source, the algorithm obtains a cut value, while ensuring that the cut splits the graph in two.

1. Create an array called on OnSourceSide with the length of the number of nodes within the graph, with an empty string at each index, with the boolean True at the index correlating to the starting node, (source/supersource). This allows the ability to see what has been assigned based on previous nodes and which hasn’t.
2. By using the breadth first search algorithm, find the outgoing edges for the current node and add the node in which the edge is going to, to the queue, to be inspected as the next current node, starting with the supersource/source. Store the nodes found in an array nodes, with their node ID’s, which will correlate to a boolean value in the array OnSourceSide. The order in which the nodes are inspected, will dictate the indexes for the correlation, having a direct correlation.
3. For each outgoing edge found in the array nodes, check if there is a collision with any of the segments of the cut.

* If there is a collision and the node which the edge is going to has already been assigned as to whether it falls on the source side or sink side, if the boolean value at the index corresponding to the node within OnSourceSide does match the value of the current node’s boolean value at its own index, the cut is valid, else the cut is invalid.
* Store the edge as a collided edge for that cut.
* Update the index of the node which the outgoing edge is going to, setting it to the opposite of the Boolean value for the current node.
* If there is no collision and the node which the edge is going to has already been assigned as to whether it falls on the source side or sink side, if the boolean value at the corresponding index within OnSourceSide does not match the value of the current node’s boolean value at its corresponding index, the cut is invalid.
* A group of colorful text boxes

  Description automatically generated with medium confidenceUpdate the index of the node which the outgoing edge is going to, setting it to the same boolean value for the current node in OnSourceSide if no collisions and the opposite boolean value if there was a collision.

1. Iterate through each element of OnSourceSide, if the element is true then add the node in correlation with the current index of the element to be on the source side, else add it to the sink side.

* Each node will have an index based on the order in which it was visited, this will match the index for nodes within OnSourceSide.

1. Check if there is at least a single collided edge and compare the nodes on either side, ensuring all the sources are on the source side and all the sinks are on the sink side. If one of these statements is false, the cut is invalid. If both of these statements are true, then:

* A white text with black text

  Description automatically generatedIterate through each edge of the edges collided with cut. If the node which the edge comes from is on the source side, add the maximum capacity of that edge to the total cut value. If the node which the edge comes from is on the sink side, reduce the total cut value by the minimum capacity of that edge.

1. Data structures used and why:

Queues are used primarily in the breadth first search algorithm. By utilising its FIFO (first in first out) structure, this enables the connections of each node to be sorted in an order from sink to source, only moving onto the next set of connections once the initial nodes have been viewed. This therefore creates a particularly useful order when determining if minimum capacities of some edges may need to rise due to the interconnected graph. Due to this being a dynamic data structure, this enables the versatility of graphs with varying numbers of nodes to be more space efficient as memory usage will vary depending on the number of outgoing edges a node has.

Dictionaries will be primarily used as references between node ID and the index at which they are stored in the list through the use of a key value pair, specifically when identifying node position in the graph compared to the cuts. This means this is time efficient as this enables quick access to specific nodes without having to sort through the entire node list. Furthermore, this is also more memory efficient than duplicating the arrays as a reference means less instances of the same object stored, reducing the duplication of data. This will be a dynamic data structure due to the number of the key value pairs being dependent on the number of nodes, which will vary between graphs.

Tuples are primarily used to put colours onto the screen as parts of the hexadecimal rgb colour as well as using coordinates to position all shapes from borders to buttons. Due to these values being fixed, this means this can’t be modified, allowing shapes to not be accidentally changed. This is a static data structure due to the tuple being of fixed length. For example, coordinates are of length two, (x, y) and colours are of length 3, (red, green, blue).

Arrays will be used the greatest, enabling composition to occur by storing objects such as a list of nodes, edges or even the paths for the maximum flow. By incorporating the use of dictionaries, this increases the time efficiency due to the searching algorithms such as linear search only must be done once, reducing the duplication of steps. However, the use of arrays allows similar objects or elements to be stored together, particularly useful for the adjacency lists, mapping each connection between the current and adjacent nodes. Due to these use cases being highly dependent on the data inputted for nodes and edges, this is a dynamic data structure as a result with the relationship changing depending on where the user is within the program.

1. Library – tkinter vs pygame:

For the system, I had 2 possible options, tkinter or pygame.

Tkinter uses inbuilt widgets to make the graphical user interface (GUI) simpler, abstracting away the functionality of buttons and scroll bars and building windows. This would allow a greater focus on the maximum flow – minimum cut theorem, combined with the use of widgets and the simpler syntax, this enables an easier to use interface for the user. This is important when traversing the application in front of a class of students so the problems are easy to set up and solve. However, in comparison to pygame, where inbuilt functions must be created from the ground up, allows the opportunity for the GUI to be much more customisable, allowing a matching aesthetic to the window.

Pygame, due to it primarily being a 2D game development platform, means this is much more interactive, allowing greater user input, potentially allowing minimum cuts to be drawn directly onto the graph while the program is running. This runs adjacent to the idea of this being a teaching tool as this allows greater interaction with the application allowing a greater variety of points to be made, coinciding with the points made. However due to tkinter not being interactable, this means this interactivity will have to come from the widgets and program itself, meaning a more general solution will be created.

For structuring the code, both libraries support object orientated approaches and the modular approach, while enabling the GUI and processes to remain separate. Pygame allows a greater use of inheritance when drawing key functionality tools such as borders and buttons.

In terms of experience with both libraries, I have used pygame much more extensively, meaning I have a greater overview of its limitations and the understanding of what general functionality I would need to add as well as the advantages of its interactive nature when running due to its refresh rate. However, on the other hand, my experience with tkinter is more general GUI’s, focusing on the visualisation elements rather than the interactive nature of various other widgets, allowing a greater focus on the teaching tool.

The system will be developed in pygame, as this enables a greater opportunity for the teacher to interact with the application making cuts more specific to the teaching point. This will enable greater interactivity while still having the ability to create a GUI which will be clear to the user, using the same functionality, except without those functions being inbuilt.

1. Use of Libraries: pygame, math:

Pygame will be the dominant library in which enables the 2D application to be displayed on the screen, with easy positional placements. From the graph to the user inputs, this enables a more customisable GUI to be made which can be aimed to be simplistic. Due to the refresh rate of the screen, this also allows greater user interactivity with the screen, particularly due to the ability to create their own cuts. This will engage possible participation, allowing key points to be found when solving the problem itself, while collision detection enables identification of intersecting segments of each cut with edges of the graph.

Math is also a library used, specifically to convert between radians and degrees as well as using trig, particularly due to the finding the angles and distances of edges and segments within each cut. This enables rotations and dynamic edge placement around the node using sin and cos due to its circular shape.

1. Modules:

My file structure will be created primarily in 4 separate files, one which has my main program focusing on the network flow elements, one will focus on general components added such as Queues, another will focus on graph elements such as nodes or edges and the other focusing on the core functionality for the user interface such as buttons and scrollbars. This means the latter 3 files will be imported into the main program, allowing utilisation of widget style features to enable the user interface to be more interactable as well as data structures such as queues and even node and edges.