DSA ASSIGNMENT REPORT

CURTIN UNIVERSITY
COMP1002
DATA STRUCTURES AND ALGORITHMS

AUTHOR: ALASTAIR KHO
STUDENT ID: 20214878
DATE: 17/10/2021

Background

This report presents the reasonings behind the implementations and justifications of use of particular data structures and algorithms in building an application that reads input text files to construct a graph-based world. The task involves to produce a system that is able to read a graph and generate all routes for the input graph from a “start” node to a “target” node. All routes must then be ranked in order of “weight”, wherein a smaller weight is considered a shorter path than a larger weight. This required implementations of several ADT’s (Abstract Data Types) to be able to construct the “graph” world, adjust parameters of the input data, generate routes in the graph, as well as sorting the routes. In addition to that, efficiency in terms of time complexity (execution times) and space complexity (memory) are also an important factor to be considered heavily for the implementations of particular ADTs. This report aims to explore the reasoning and justifications behind favouring particular ADTs for the task in constructing the graph-routing system, such as Min-Oriented Heaps, Hash Tables, Linked Lists, Graphs and Queues.

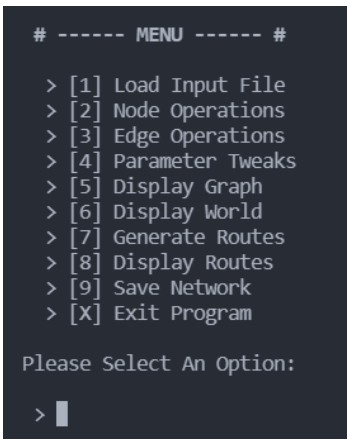
User Guide

The python code “gameofcatz.py” is the main file used for running and executing the menu program. Python 3 is the Python version required to run this program. Before running this file, there are external modules/packages that are required to be pre-installed for the program to function. These includes;

- **Matplotlib**: This is a module used primary for plotting and presenting graphs.
- **Numpy**: This is an arithmetic module that provides the required mathematical functions and the array data structure.
- **NetworkX**: This module provides a system in producing visualisations of the Graph abstract data type. NetworkX works with the Matplotlib module in order to output a visual form of a procedurally coded Graph data structure.

To run the program, the python3 command needs to be executed with “flags” or “command-line arguments”. The program comes with two different modes; interactive mode and simulation mode.

The Interactive mode enables the user to enter a menu system with a user interface that provides options for loading compatible files, altering nodes and edges, obtaining data about the nodes and edges, adjust parameters for the simulation such as start and target nodes, previewing the graph’s weighted adjacency matrix, creating a visual representation of the graph, generating and displaying routes, as well as saving files (with any changes made from the first few options in the menu). To run in interactive mode, the “-i” flag must be used. Please enter the following command in the terminal to enter interactive mode:



python3 gameofcatz.py -i

This will output a menu that you can choose from to interact with the graph.

```
Please Select An Option:

 > 1
   |    > Load Input File <
   |    ----------------------------
   |
   |    Please enter file name WITH extension: input.txt
```

Before any of the other options (except exiting program) can be used, an input file must be supplied. Select the first option (by inputting “1”), then enter the file name to read the Graph data. Note that the input file must be compatible with the program, the program will only be able to read data it recognises. Once the input file has been supplied, all of the other options become available.

```
Please Select An Option:

 > 2
   |    > Node Operations <
   |    ----------------------------
   |
   |    Enter node operation functions to run.
   |    i.e. addVertex(5 , 1)
   |    Note that deleting/adding nodes may also alter edges. Available node operations:
   |    getVertex(label), hasVertex(label), removeVertex(label)
   |    getVertexCount(), displayAsList(), addVertex(label, weightCode)
   |    updateCode(node, newCode)
   |    The following are node types (for adding vertices) presented in the format: (Code, Weight)
   |
   |    Node Types: (Code: B, Weight: 3), (Code: -, Weight: 0), (Code: D, Weight: 100), (Code: F, Weight: -3), (Code: H, Weight: 2), (Code: T, Weight: 1),
   |    Enter node operation functions to run. Call exit() to return to menu.
   |
   |    # ---  Pseudo Interpreter  --- #
   |
   |    >>>
```

The second and third option provides accessibility to data and interaction with the graph. This includes getting vertex/edge data, and updating vertex/edge data. These are to be executed by calling “pseudo-functions” in the input that is in a manner similar to the python3 interactive command interpreter. This built in “pseudo interpreter” is more restrictive than the python3 command interpreter as it always assumes that inputs are strings (conversions to other data types are done in the background within the application), hence entering the commands: “addVertex(Hello, F)” and treats the arguments “Hello” and “F” automatically as strings (similar to BASH).

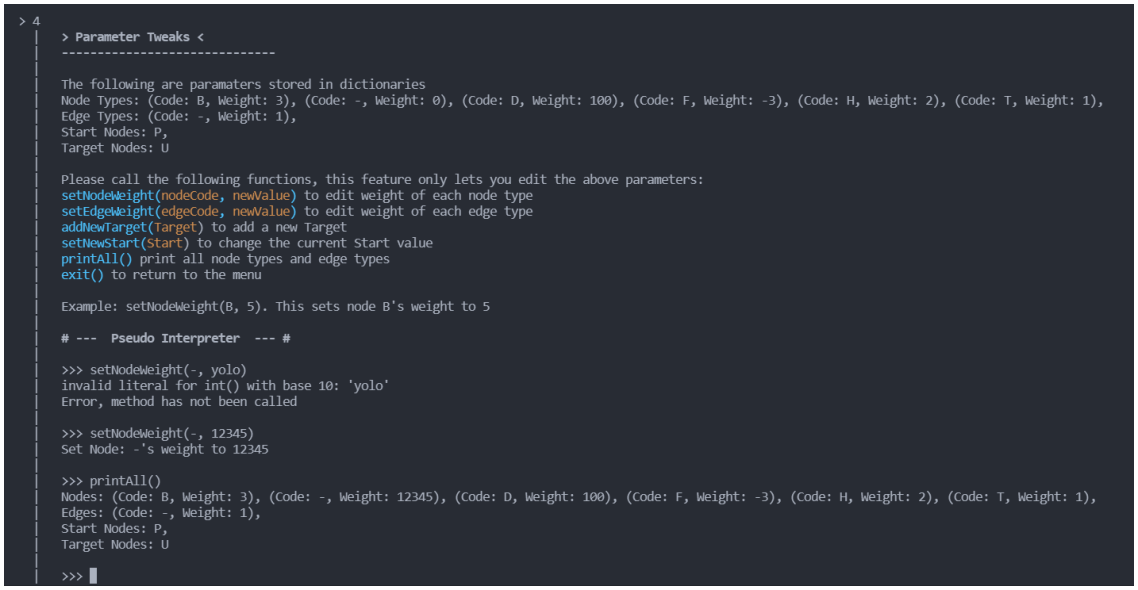
Example:

```
   |    > Node Operations <
   |    ----------------------------
   |
   |    Enter node operation functions to run.
   |    i.e. addVertex(5 , 1)
   |    Note that deleting/adding nodes may also alter edges. Available node operations:
   |    getVertex(label), hasVertex(label), removeVertex(label)
   |    getVertexCount(), displayAsList(), addVertex(label, weightCode)
   |    updateCode(node, newCode)
   |    The following are node types (for adding vertices) presented in the format: (Code, Weight)
   |
   |    Node Types: (Code: B, Weight: 3), (Code: -, Weight: 0), (Code: D, Weight: 100), (Code: F, Weight: -3), (Code: H, Weight: 2), (Code: T, Weight: 1),
   |    Enter node operation functions to run. Call exit() to return to menu.
   |
   |    # ---  Pseudo Interpreter  --- #
   |
   |    >>> getVertex(A)
   |    (A, 0)
   |
   |    >>> getVertexCount()
   |    21
   |
   |    >>> removeVertex(A)
   |    (A, 0)
   |
   |    >>> addVertex(Hello, 0)
   |    None not a valid vertex type!
   |    Error, method has not been called
   |
   |    >>> addVertex(Hello, -)
   |
   |    >>>
```

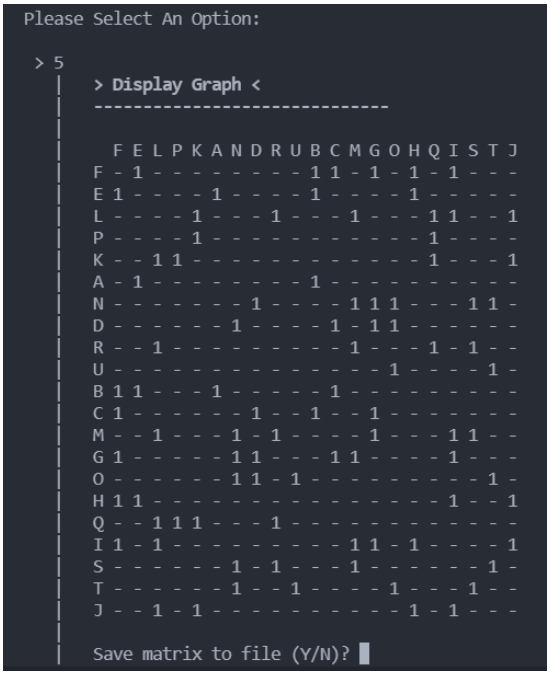
Note that available weight codes are presented in the description, these Codes (“B”, “-“, and others) must be used instead of the numeric value. To exit from the pseudo interpreter, enter the “exit()” function. This pseudo-interpreter invites for flexibility, ease of access and (and potentially familiarity for the user if the user understands programming languages).

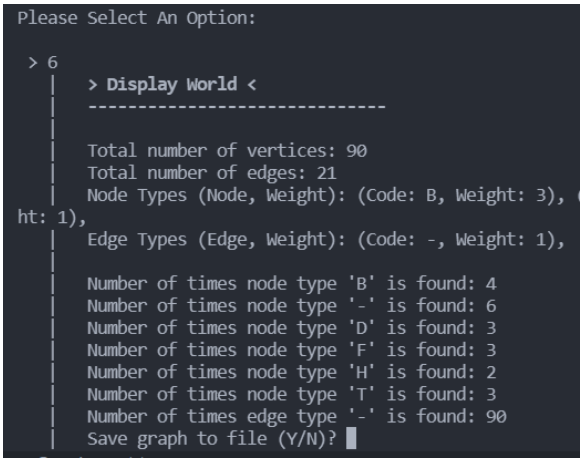
The parameter tweaks section provides a similar “pseudo-interpreter” setup. To edit the node type’s corresponding weight, type in “setNodeWeight(nodeCode, newValue)”.

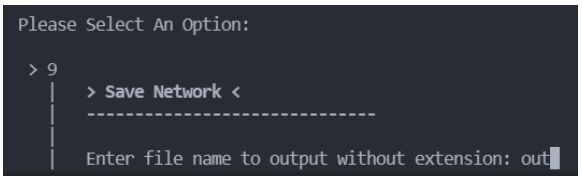
Example:

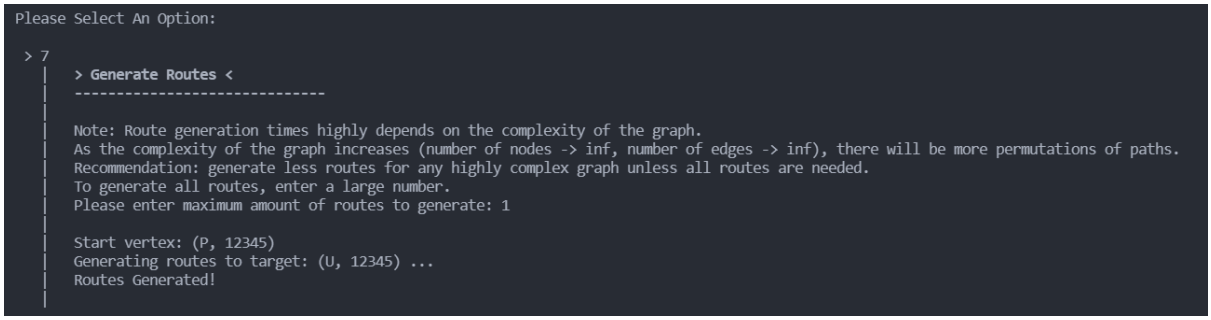


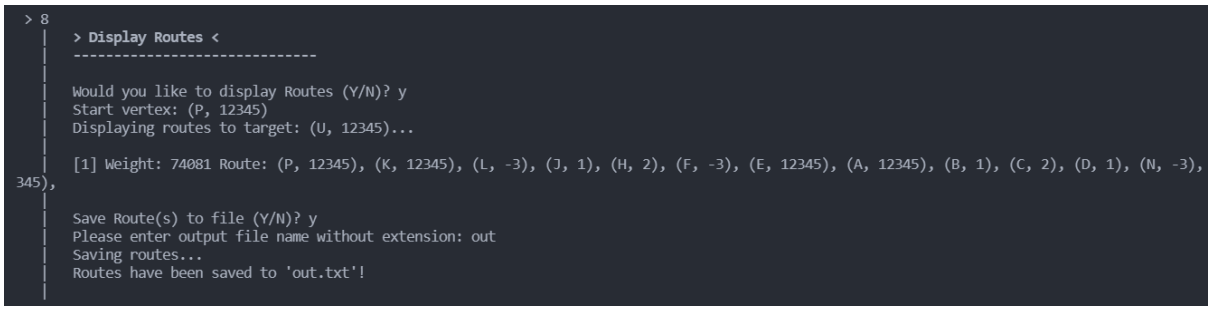
The other options provided in the menu (of which include displaying the matrix and graph, generating and displaying routes, as well as saving network) have a more straightforward experience as the other options do not require as much user input.











The 9th option which is the “Save Input” option allows the user to save the current graph (with the changes made in the program) in the same format as the input file. This allows the user to read the output file again with the program even after closing the previous program to continue making alterations to the graph.

This menu system with the pseudo-interpreter is designed to provide flexibility, familiarity and ease of access to data for its user. The aim for this design is an attempt to reduce clutter in terms of large outputs and printing (which may cause the user to lose track of the data) as well as repetitive input demands (such as input statements that ask the user if they wish to repeat the system).

Description of Classes

This program comes with 5 classes of which all are of importance to the functionality of the program. These classes includes: Graph class, Linked List Class, Hash Table, Min-Based Heap and Queue Classes.

Directed Graph Class (Previously submitted in Prac 6, modified in the program)

The Directed Graph class is essential to the program as it is the core component in the task that provides graph generation as well as graph routing generation. This graph class enables the program to keep track of vertices, edges as well as performing operations and calculations on the vertices and edges and obtaining data from it. This graph class is used in the gameofcatz.py program to retrieve matrix form of the graph, keeping track of each vertex, edges and its interconnection, as well as generate routes that is based off the depth first search algorithm. The graph class is favoured over other classes such as trees or dictionaries as its structure (in terms of connection between other objects such as vertices and edges) is more suited for creating a system of networks. This offers

easier of access in terms of obtaining data about a particular vertex and edge, and its adjacent objects. This works well with route generation as well as performing traversals becomes more feasible to keep track of connected vertices through an adjacency list (heap array structure used instead of a “list” in the program). A tree structure, although is a type of graph structure, would not be suitable as a tree structure is limited in a sense that its structure is confined to a hierarchical system (parent greater than child system). The graph class is more appropriate as it provides more flexibility in accessing connections between vertices and edges as well as holding each corresponding “weights”, of which hold high importance to calculating the best and least weighted routes.

Linked List Class and Queue Derivative Class (Previously submitted in Prac 4, modified in the program)

Linked List and queue class provides a sequence of values interconnected in order not by placement in memory but through references to each other, wherein each “list node” holds the object value and a reference or pointer to the next value in sequence as well as the previous value. This results in a theoretically infinite, interconnected sequence of objects that can be stored in a linked list. This can be favoured over arrays as arrays are more restrictive in a sense that it requires pre-allocation of memory and thus cannot change in length. Linked lists provide more freedom as the theoretical length of a linked list is infinite (but realistically limited to maximum memory available) and it also allows for multiple data types (where as arrays is limited to one data type). Although the space complexity of linked lists is O(n) when compared to arrays which are O(1), this program may generate a multitude of routes with differing input files and the route generation amount is generally unknown and therefore cannot be assigned to an array to a predefined space. Linked Lists insertion and deletion may be faster than other implementations as it only involves creating an object in the memory space, and then linking it with pointers/references. The decision to favour a linked list over arrays is primarily for the linked list’s ability to dynamically insert new objects and its theoretical infinite length. This is implemented in the Graph route generation feature to store a chain of vertices as routes, as there are many differing routes (ranging from large to small) with a multitude of permutations of vertices a graph that can generated (more vertices to traverse). As the sizes of routes in a graph extend longer, arrays can become less favourable due to its fixed length nature as linked lists can dynamically change size, whereas demanding memory allocation for a large array but only inserting a small route (composed of little vertices) becomes a waste of memory allocation. This justification also applies for the queue derivative and the queue version can also be favoured due to its FIFO nature.

Hash Table Class (Previously submitted in Prac 7)

The Hash Table class offers an ease of access for a dictionary like structure. The Hash Table holds key and value based elements of which accessing a value with its corresponding key becomes an O(1) operation (on average) due to the data being held in an array and a system of hashing a key to obtain it’s corresponding index value. Although arrays are of a fixed length structure, the hash table has an array system that can dynamically “resize” its length. This “resize” implementation this holds an O(n) time complexity. The dynamic ability of the hash dictionary is more desirable for this program as the input files for the program could have a multitude of vertices or edges and differing types of which may require a key to access its value. Furthermore, holding in a fixed allocated array

may result in not enough spaces to add new edges or vertex types. An insert, delete and look up of hash table generally averages a time complexity of O(1) with a worst case of O(n). This is more favourable than holding values in a linked list as iterating over a linked list to find the desired node is generally O(n). Although accessing elements in a fixed array holds a time complexity of O(1) , the fixed array can't dynamically change it's length nor can it hold key to value pairs.

Min-Based Heap Array (Previously submitted in Prac 8 as max-oriented, modified to a min heap)

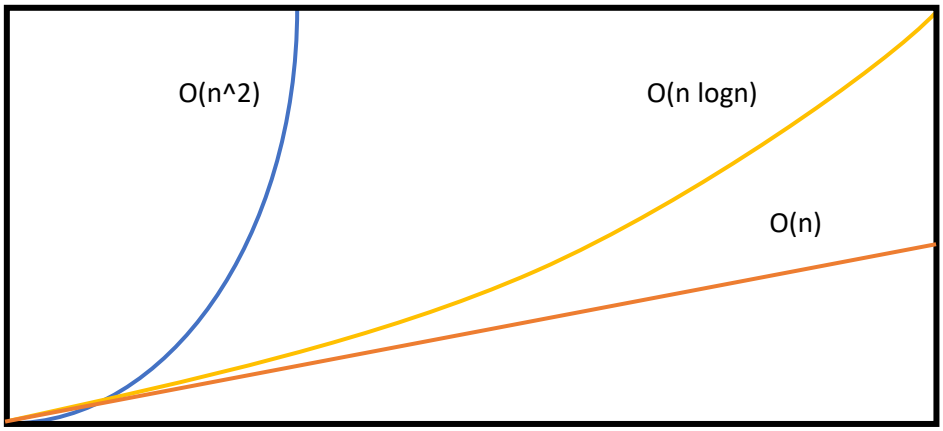
The min-based heap holds values in an array structure that allows for a priority-based ordering. As a part of the task requirements is to generate routes and rank and sort them according to its accumulated weight, holding values in a minimum-oriented heap becomes more favourable than holding values in a linked list when priority or order matters in the assortment. The min-based heap follows a tree like structure that is implemented in the form of an array, of which the parent priority must always be smaller than the child's priority. The heap maintains this order whilst deleting and inserting and the elements are held in an array, therefore deletion and insertion in a heap becomes an O(log n) time complexity operation. This can be more favourable than linked lists or arrays for this graph structure as the route generation and sorting routes based on weights becomes easier to access and can be faster. This min-based heap is implemented in the Graph class in adjacency lists, edge and vertices lists as well as route lists. A min-based heap in these situations is favourable for generating the best routes when lower number of routes is generated for faster run times. This is because the route depth first search method will always iterate through an ascending order of adjacent vertices (and prioritise a smaller weight or priority). Therefore generating 10 routes out of a possible 10000 routes will almost guarantee the top best routes on average. In addition to that, the sorting algorithm that is supplied with the min-based heap structure aids with sorting the generated routes even faster compared other sorting algorithms (sorting based on total weights in each route to rank each of them). The heap sort in the heap class holds a time complexity O(n log n) sorting algorithm of which is significantly better than other sorting algorithms such as bubble sort, insertion sort and selection sort of which hold time complexities between O(n^2) and O(n). This makes the min-based heap more favourable than other implementations.

Justification of Decisions

Sorting Algorithm and Heap structure

The justification behind using a heap sort instead of bubble, insertion and selection sort, and a heap like structure over other ADTs such as a generic linked lists or array is that sorting and ranking (where sorting by weight becomes important) becomes faster with a time complexity that is logarithmic. With a heap structure, insertion and deletion hold an O(log n) time complexity. As other sorting algorithms may produce worst case results of O(n^2), O(n log n) sorting algorithms are more favourable due to it's nature being closer to a linear time complexity. As the data size or the input size grows or an increase in space complexity, O(n log n) becomes especially more favourable than O(n^2) sorting algorithm times as the O(n log n) sorting algorithm correlates more closely to a constant time complexity over time(but not truly constant). Other sorting algorithms such as bubble sort, insertion sort and selection sort have best cases and worst cases ranging from O(n) and O(n^2),

making them highly unfavourable for sorting large amounts of data such as the number of routes for a highly complex graph in ascending order.



Parsing, reading, sorting and interpreting big data in the real world makes both time and space complexity and important factor to consider as efficiency and efficacy becomes of high importance and priority. Reducing these operation times on big data allows for faster interpretation and understanding of data. Thus, the decision for a heap structure and heap sorting algorithm for this task is significant in a sense that it reduces the sorting time for ranking routes, and a for heap structure; reducing its insertion and deletion times. These time and space complexities becomes of extreme importance when trying to interpret large amounts of data, and thus the decision for a logarithmic based algorithms is highly favourable.

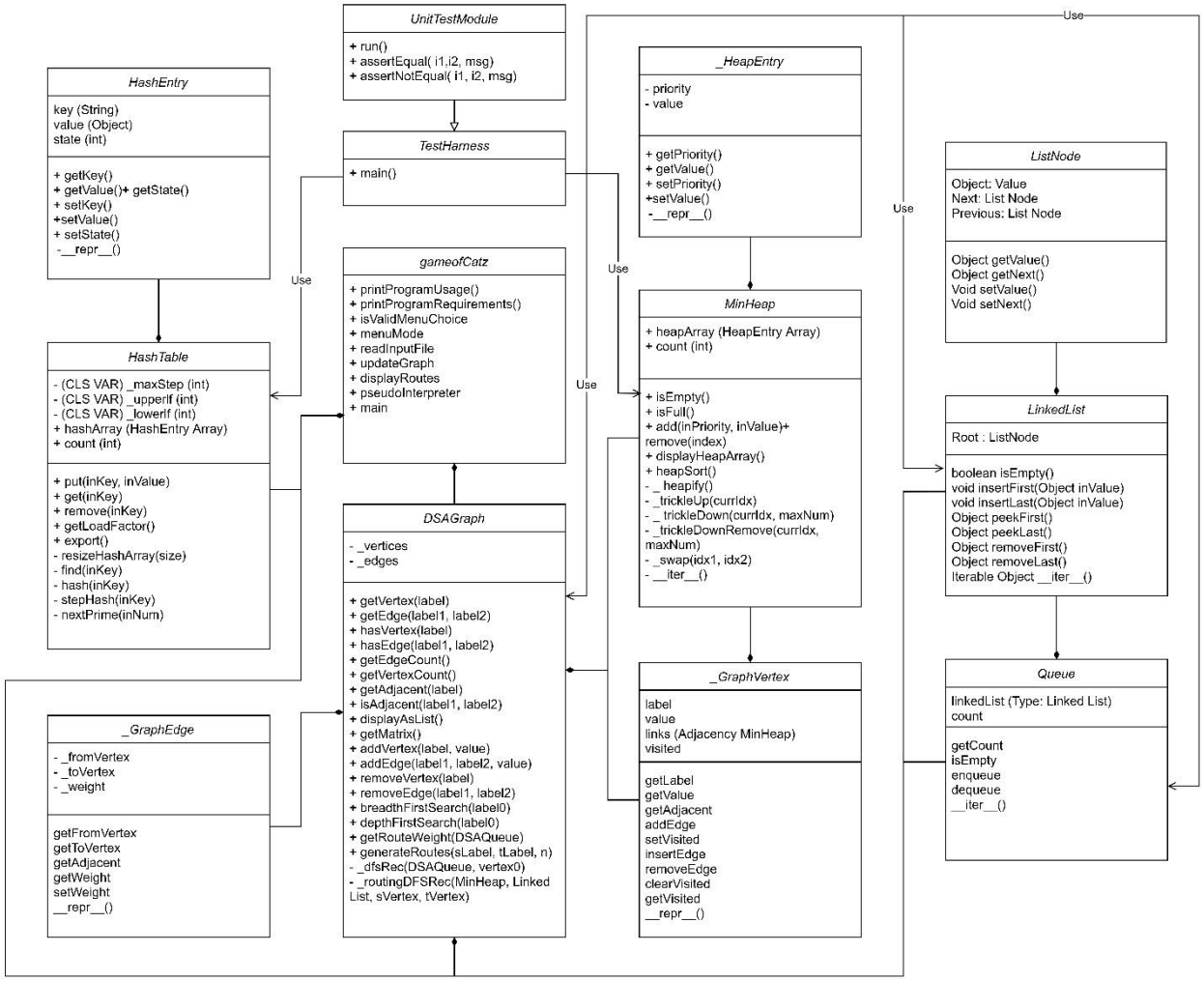
A Pseudo-Interpreter in the Menu System (Borrowed and modified from TestHarness.py in Prac 5)

The decision to produce a Pseudo-Interpreter was made to make interacting with the Graph abstract data type more easily, efficiently and familiar. The Pseudo - interpreter is designed to hold similarities with the current built in python terminal interactive mode for familiarity as well as reducing output and prompt cluttering. In most menu systems, prompts may appear several times (with some repeating with a yes or no input to repeat the process at the end). This becomes somewhat tedious as the amount of output prints becomes cluttered with older prints moving to the top as well as input prompts in the UI which can become repetitive. This pseudo-interpreter aims to reduce repetitive input prompts and print clutters with a more familiar design.

Saving System

The decision to choose an output text save over a serialized save is to keep the input and output flow the same since the input appears to always be a text file. Therefore, whatever is inputted, it is outputted in the same format and can be inputted again. Furthermore, serialization may make the input and outputs more intricate as multiple saves (of node/edge dictionaries, node/edge type dictionaries and graph storages with byte files and text files) may cause the input and output file system to become confusing.

Unified Modelling Language



Traceability Matrix

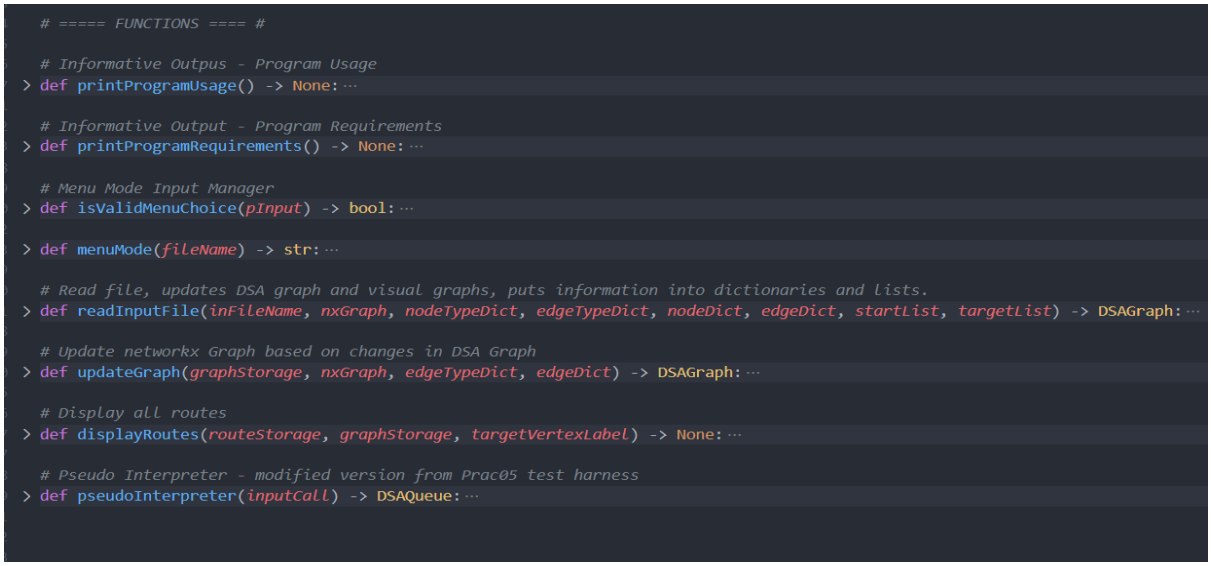
FEATURE	CODE	TEST
0 Modes and Menu	gameofcatz main()	
0.1 Interactive Mode	line 270	[PASSED] Runs as expected when entering command line argument
0.2 Simulation Mode	line 940	[PASSED] Runs as expected when entering command line argument. File is saved as expected
0.3 Usage Requirements	lines 27-37	[PASSED] Runs as expected when not entering command line argument
0.4 Menu Operations	lines 40-78	[PASSED] Menu appears as expected
1. Load Data		
1.1 Read File	lines 81-147	[PASSED] Reads menu as expected
1.2 Loads check for invalid/missing	lines 994-997, 303-316	[PASSED] Ignores invalid data, checks for missing data (achieved by remove only Ncodes for it to not know what weight each vertex holds)
1.3 Loads serialised	Not implemented	Not implemented
2 Node operations		
2.1 Pseudo-Interpreter	lines 179-220,	[PASSED] Accepts listed functions and rejects bad function calls (rejects bad functions when parenthesis is not supplied or arguments are invalid)
2.2 Add	lines 395 - 410	[PASSED] Add is functional and corresponds in graphStorage as well as local storage such as hash dictionaries/ Method: tried calling function in pseudo interpreter
2.3 Remove	lines 370-392	[PASSED] Remove is functional and corresponds in graphStorage as well as local storage such as hash dictionaries. Method: tried calling function in pseudo interpreter
2.4 Update	lines 414-430	[PASSED] Update is functional and corresponds in graphStorage as well as local storage such as hash dictionaries/ Method: tried calling function in pseudo interpreter. Method: tried calling function in pseudo interpreter
3 Edge Operations		
2.1 Pseudo-Interpreter	lines 179-220,	[PASSED] Accepts listed functions and rejects bad function calls (rejects bad functions when parenthesis is not supplied or arguments are invalid)
2.2 Add	lines 395 - 410	[PASSED] Add is functional and corresponds in graphStorage as well as local storage such as hash dictionaries. Method: tried calling function in pseudo interpreter
2.3 Remove	lines 370-392	[PASSED] Remove is functional and corresponds in graphStorage as well as local storage such as hash dictionaries. Method: tried calling function in pseudo interpreter
2.4 Update	lines 414-430	[PASSED] Update is functional and corresponds in graphStorage as well as local storage such as hash dictionaries. Method: tried calling function in pseudo interpreter
4 Parameter Tweaks	lines 579-698	[PASSED] Updates reflects both in local dictionary storages and graphStorage. Method: tried calling function in pseudo interpreter
5 DisplayGraph	lines 702-732	[PASSED] Presents matrix obtained from graphStorage, then provides option to save
6. Display World	lines 735-804	[PASSED] Presents networkx representation of graph, then provides option to save
7. Generate Routes	lines 807- 835	[PASSED] Updates routes in route storage
8 Display Routes	lines 167-177, 838-885	[PASSED] Obtains routes from storage and presents correct ranked routes
9. Save Network		
9.1 Save Text File	lines 888-932	[PASSED] Outputs in file similar format as supplied text files in assighment. Tried exiting program and reading this file again, functional as expected.
9.2 Save Serialized	Not implemented	Not implemented
10 Others		
10.1 Heap Sort Min	MinHeap.py lines 106-168	[PASSED] Used ClassTestHarness.py to test heap sorting trickle up and trickle down with assertions to each value in heap array
10.2 Graph Route Generation	DSAGraph.py lines 317-380	[PASSED] Used ClassTestHarness.py to test accurate route generations with assertions to each value in the output

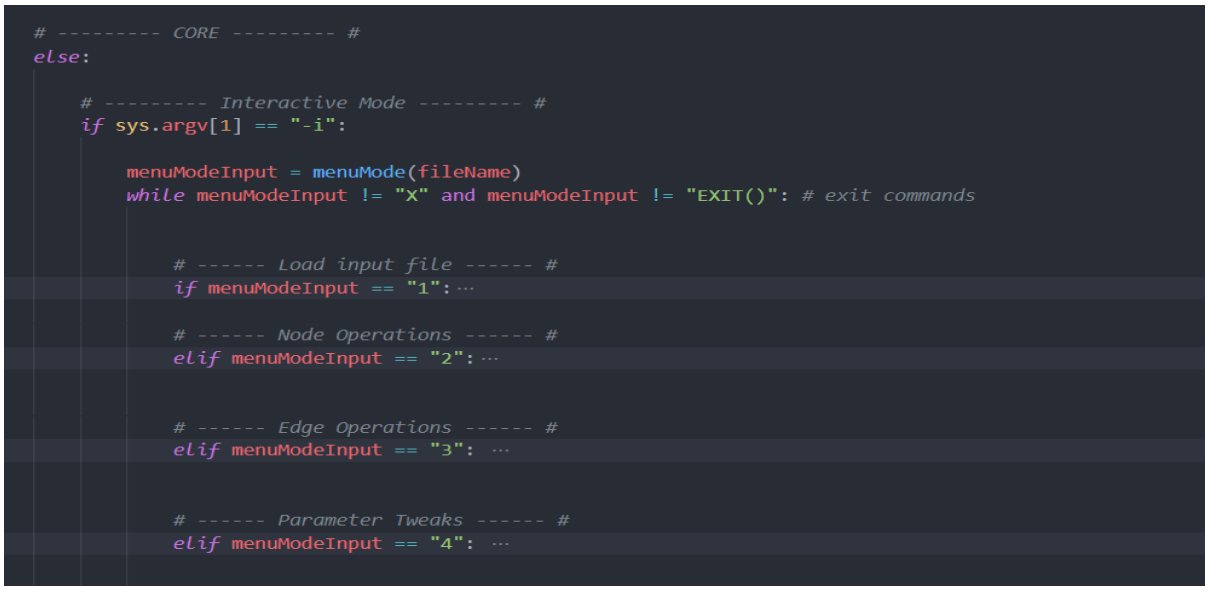
Code Showcase

The current code for the program and its relationship with the classes is structured in a similar manner to a server-client relationship, wherein the gameofcatz.py is in constant communication with classes, primarily with DSAGraph.py so that gameofcatz.py can get information from DSAGraph.py, request updates and receive updates to the Graph object. In this case, gameofcatz.py acts like a “server” that cross communicates with its own storage and the “client” DSAGraph storage (to match and validate), as well as other abstract data types such as Hash Tables and linked lists. The DSAGraph.py as well as the other classes constructs its abstract data type and is either used within another class or used within gameofcatz.py, wherein gameofcatz.py has its own local storage that stores information. Every time gameofcatz.py receives user input to change data, it has to update its

local data (node dictionary, node type dictionaries, etc..) and the data in the graph (_vertices, _edges, _links). The DSAGraph.py can then use the received information to perform algorithms such as route generation and send it back to gameofcatz.py for the user, this is similar with the other class implementations. The gameofcatz.py has its own local dictionary storage of known vertices, vertex types, edges and edge types to match edge codes and its weights with edges in the DSAGraph, this also applies to all vertices. This is because DSAGraph.py doesn't hold edge/vertex string codes as it relies on integer weights for its heap priority structure and sorting implementation. These dictionaries also hold information to update the visual network graphs and information for saving files.

The main code is structured in order of modules, functions, core components and program end. The main code includes the pseudoInterpreter() function holds the code for interpreting the pseudo interpretation feature in the second, third and fourth mode in the program. The classes hold all the algorithms that are used in this assignment.





Scenarios

Scenario 1: Standard Simulation Run With gameofcatz.txt (less complex graph)

The standard simulation run tests a functional route generation algorithm and file reading systems. In this simulation we are using gameofcatz.txt as the input file. The following command was performed, where “output” is the text file name for the output:

python3 gameofcatz.py -s gameofcatz.txt output

The following is found in the output file which holds a small section of the generated paths in a ranked order:

```
# ========== Routes From (A, 0) To (J, 0) ========== #

Rank: 1 Weight: 3 Route: (A, 0), (E, 0), (F, 0), (G, -1), (I, -1), (J, 0),
Rank: 2 Weight: 3 Route: (A, 0), (E, 0), (F, 0), (I, -1), (J, 0),
Rank: 3 Weight: 4 Route: (A, 0), (B, 1), (F, 0), (G, -1), (I, -1), (J, 0),
Rank: 4 Weight: 4 Route: (A, 0), (B, 1), (C, 0), (G, -1), (I, -1), (J, 0),
Rank: 5 Weight: 4 Route: (A, 0), (B, 1), (F, 0), (I, -1), (J, 0),
Rank: 6 Weight: 4 Route: (A, 0), (E, 0), (F, 0), (C, 0), (G, -1), (I, -1), (J, 0),
Rank: 7 Weight: 5 Route: (A, 0), (B, 1), (C, 0), (D, 0), (G, -1), (I, -1), (J, 0),
Rank: 8 Weight: 5 Route: (A, 0), (B, 1), (F, 0), (C, 0), (G, -1), (I, -1), (J, 0),
Rank: 9 Weight: 5 Route: (A, 0), (E, 0), (B, 1), (F, 0), (I, -1), (J, 0),
Rank: 10 Weight: 5 Route: (A, 0), (B, 1), (E, 0), (F, 0), (I, -1), (J, 0),
Rank: 11 Weight: 5 Route: (A, 0), (E, 0), (B, 1), (F, 0), (G, -1), (I, -1), (J, 0),
Rank: 12 Weight: 5 Route: (A, 0), (E, 0), (B, 1), (C, 0), (G, -1), (I, -1), (J, 0),
Rank: 13 Weight: 5 Route: (A, 0), (B, 1), (C, 0), (F, 0), (G, -1), (I, -1), (J, 0),
Rank: 14 Weight: 5 Route: (A, 0), (B, 1), (C, 0), (G, -1), (F, 0), (I, -1), (J, 0),
Rank: 15 Weight: 5 Route: (A, 0), (B, 1), (E, 0), (F, 0), (G, -1), (I, -1), (J, 0),
Rank: 16 Weight: 5 Route: (A, 0), (E, 0), (F, 0), (C, 0), (D, 0), (G, -1), (I, -1), (J, 0),
Rank: 17 Weight: 5 Route: (A, 0), (B, 1), (C, 0), (F, 0), (I, -1), (J, 0),
```

This presents a functional graph route generation algorithm with a functional sorting algorithm. The time taken for the program to finish was 3.88 seconds. This is primarily due to the small number of routes (108 routes produced) that were possible as the input graph. The maximum amount of routes defaulted is 1 000 000 000, this can be changed in gameofcatz.py under settings, this number may be required to be reduced if your machine does not have sufficient RAM.

Scenario 2: Standard Simulation Run With gameofcatz2.txt (highly complex graph)

In this simulation we are using gameofcatz2.txt as the input file. The following command was performed, where “output” is the text file name for the output:

python3 gameofcatz.py -s gameofcatz2.txt output

The following is found in the output file which holds a small section of the generated paths in a ranked order:

```
# ========== Routes From (P, 0) To (U, 0) ========== #

Rank: 1 Weight: 7 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 2 Weight: 7 Route: (P, 0), (K, 0), (L, -3), (R, 3), (S, 3), (N, -3), (O, 0), (U, 0),
Rank: 3 Weight: 8 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (E, 0), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 4 Weight: 9 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (B, 1), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 5 Weight: 9 Route: (P, 0), (K, 0), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 6 Weight: 9 Route: (P, 0), (K, 0), (J, 1), (L, -3), (R, 3), (S, 3), (N, -3), (O, 0), (U, 0),
Rank: 7 Weight: 9 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (O, 0), (U, 0),
Rank: 8 Weight: 9 Route: (P, 0), (K, 0), (L, -3), (R, 3), (S, 3), (N, -3), (D, 1), (O, 0), (U, 0),
Rank: 9 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (R, 3), (S, 3), (N, -3), (T, 3), (U, 0),
Rank: 10 Weight: 10 Route: (P, 0), (Q, 3), (L, -3), (R, 3), (S, 3), (N, -3), (O, 0), (U, 0),
Rank: 11 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (E, 0), (F, -3), (B, 1), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 12 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (N, -3), (T, 3), (U, 0),
Rank: 13 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (E, 0), (F, -3), (C, 2), (D, 1), (O, 0), (U, 0),
Rank: 14 Weight: 10 Route: (P, 0), (K, 0), (J, 1), (H, 2), (E, 0), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 15 Weight: 10 Route: (P, 0), (Q, 3), (L, -3), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 16 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (E, 0), (B, 1), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 17 Weight: 10 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (E, 0), (B, 1), (C, 2), (D, 1), (N, -3), (O, 0), (U, 0),
Rank: 18 Weight: 11 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (E, 0), (F, -3), (C, 2), (D, 1), (N, -3), (T, 3), (U, 0),
Rank: 19 Weight: 11 Route: (P, 0), (K, 0), (L, -3), (Q, 3), (R, 3), (S, 3), (N, -3), (O, 0), (U, 0),
Rank: 20 Weight: 11 Route: (P, 0), (K, 0), (L, -3), (J, 1), (H, 2), (F, -3), (C, 2), (D, 1), (N, -3), (O, 0), (T, 3), (U, 0),
```

The time taken for the program to finish was 1 minute and 53 seconds. This was due to the larger number of routes being generated (168724 routes in total) as the input graph is a highly complex graph with a multitude of permutations of routes. If a sorting algorithm that experiences O(n^2) time complexity was used, this may have extended to even longer times for the program to end.

Scenario 3: Interactive Run With gameofcatz3.txt (high complexity graph)

In this scenario, we will test the functionalities of the interactive mode. To enter interactive mode, enter the following:

python3 gameofcatz.py -i

Once you are in the menu system, load the gameofcatz3.txt input files and follow the instructions provided in the menu system. Once the file has been loaded, enter the Node Operations Mode and enter the following commands:

```
# ---  Pseudo Interpreter  --- #

>>> addVertex(NewVertex, -)

>>> exit()
Exiting pseudo-interpreter for node operations!
```

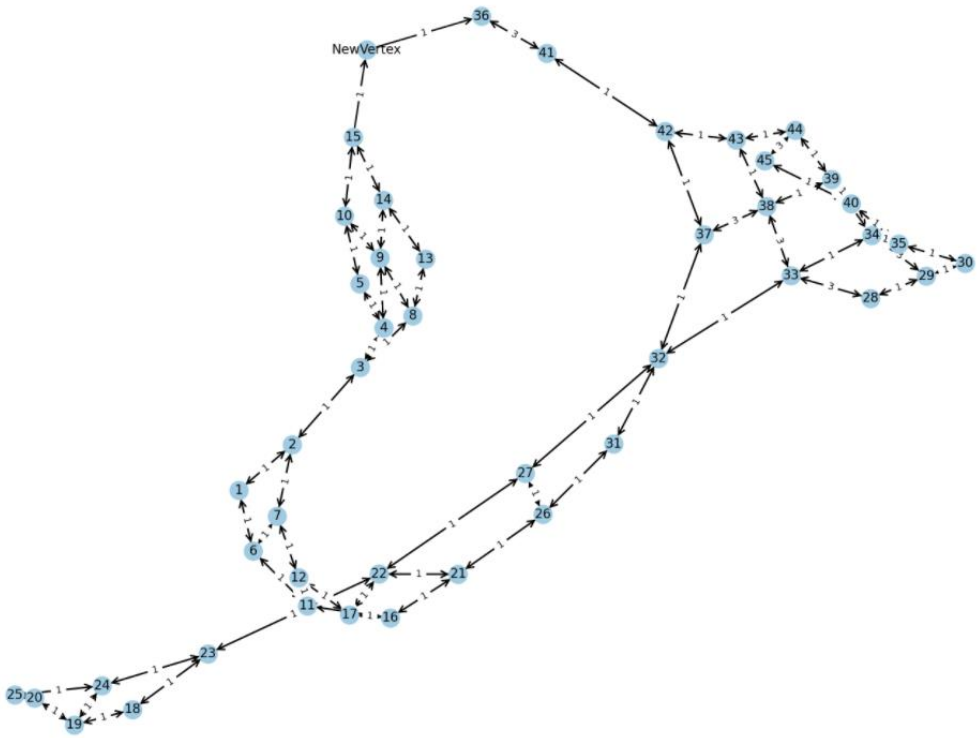
Exit the node operations menu (by calling exit()) and enter the edge operations menu. Since the start vertex in the input graph is 15 and a target is 36, we will be attempting to create a shortcut between 15 and 36 using the vertex we have created above. You can see the inputs being acknowledged by the Graph ADT by checking with the functions “hasVertex(inLabel)” or ”hasEdge(fromLabel, toLabel)” or “printAsList()”. These methods (along with addVertex(label, weightCode) and addEdge(fromLabel, toLabel, weightCode))will directly

interact with the Graph class ADT. Note that adding an edge creates a directional edge, not a multidirectional edge, to create a multiple directional edge, a directional edge needs to and from each input labels.

To create the shortcut from start to end, enter the following commands in the Edge Operations Mode:

```
# ---  Pseudo Interpreter  --- #

>>> addEdge(15, NewVertex, -)

>>> addEdge(NewVertex, 36, -)

>>> hasEdge(15, NewVertex)
True

>>> hasEdge(NewVertex, 36)
True

>>> displayAsList()
```

Now exit from node operations using the exit() command. We can check if the graph has created the connection by entering the Display World mode which provides a visualisation of the graph using the module NetworkX and matplotlib.



To fully validate that a direction edge has been made from node 15, to “NewVertex” to node 36, we can test this by using the route generation feature. First use option 7 to generate routes, then enter 10 maximum routes to generate (although there are more than 10 possible routes, the sorting algorithms should try to prioritize this shortcut vertex sequence).

```
Please Select An Option:

 > 7
   |   > Generate Routes <
   |   ----------------------------
   |
   |   Note: Route generation times highly depends on the complexity of the graph.
   |   As the complexity of the graph increases (number of nodes -> inf, number of edges -> inf), there will be more permutations of paths.
   |   Recommendation: generate less routes for any highly complex graph unless all routes are needed.
   |   To generate all routes, enter a large number.
   |   Please enter maximum amount of routes to generate: 10
```

Once the routes have been generated, enter the display routes mode by entering 8 in the main menu. Type in “Y” to display routes, then enter 36 to display the routes to node 36.

```
Please Select An Option:

 > 8
   |   > Display Routes <
   |   ----------------------------
   |
   |   Would you like to display Routes (Y/N)? y
   |   Start vertex: (15, 4)
   |
   |   Multiple Targets Detected!
   |   Target: 36
   |   Target: 40
   |   Please select the target for the route (case sensitive): 36
```

The first entry should be the following sequence of vertices: (15, 4), (NewVertex, 0) (36, -5)

```
1   Rank: 1 Weight: 1 Route: (15, 4), (NewVertex, 0), (36, -5),
2   Rank: 2 Weight: 40 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
3   Rank: 3 Weight: 41 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
4   Rank: 4 Weight: 42 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
5   Rank: 5 Weight: 45 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
6   Rank: 6 Weight: 45 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
7   Rank: 7 Weight: 47 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
8   Rank: 8 Weight: 50 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
9   Rank: 9 Weight: 55 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
10  Rank: 10 Weight: 57 Route: (15, 4), (10, 4), (9, 0), (4, 2), (3, 0), (2, 0), (1, 0), (6, 0), (7, 0), (12, 0), (17, 0), (16, 0), (21, 0),
11
```

Once you have generated the routes, enter option 9 to save the updated network. After the network has been saved, you may exit the program and attempt to read the saved network file, this should provide the same graph structure as the saved network graph structure.

Conclusion

This program aims to provide the user to “explore” the supplied input graph file and to find routes from start and target nodes. The task involves creating graph-based menu system that can read input files and present/modify its information. The program gameofcatz.py also includes implementations of ADTs such as Graphs, Linked Lists, Hash Tables, Min-Based Heaps and Queues. The justification behind choosing these abstract data types and the heap sort sorting algorithm was to increase efficiency and efficacy in terms of time and space complexities (such as heap array’s O(log n) time complexities and heapsort for its O(n log n) time complexity) as these become of high importance when reading through large amounts of data (such as the large amount of permutations of routes from a start to a target node). However, this structure could be improved by trying to implementing better, faster and more efficient sorting algorithms such as quick sort instead of heap sort as quick sort (with an average of O(n log n)), can be faster than heap sort. In addition to this, a transition to more reliance on arrays rather than linked lists could possibly mean faster route generation and sorting times. The current implementation with the pseudo-interpreter intends to enable users to access and modify the graph more easily, however this setup may not be familiar to not programmers using the program, a simpler option may be provided instead for unfamiliar users. In addition to this, more features such as reading serialized files and text files, reducing the amount of abstract data types used (to reduce memory usage) as well as implementing a game functionality are amongst the list of possible improvements to this program. One interesting improvement that could be included is the implementation of Artificial Intelligence and Machine Learning systems which may be able to generate the best routes more efficiently.