1. User Guide

The basic purpose of the code is to find out all possible paths from a ‘start’ point to ‘end’ point and rank those paths on basis of some scheme. The code sets up a world using one of the text files available in the folder. The world has different kind of nodes, and those nodes relate to each other through different kind of edges. One of the nodes is marked as ‘Start’ and one is marked as ‘Target’.

The main file that user needs to run is gameofcatz.py. There are three ways to run this file. All three ways give different results.

1. **pyhton3 gameofcatz.py**

When the file is run from command line without any arguments, it prints some basic information about the code and also let the user know about other ways to run the file

2. **python3 gameofcatz.py -i**

When the code is run with ‘-i’ as a command line argument, the code enters interactive testing environment. In this environment, user can perform several operations on the world. Available operations include, loading world from an input file, node operations, edge operations, parameter tweaks, operations to display graph, create routes and display routes, rank routes and save back to file yours.”

3. **python3 -m infile outfiles**

When the code is run like this, it creates a graph using infile and saves all paths in a prescribed order in outfile.

2. Description of Classes

Game()

Game class is the main class. It is saved in game.py. The gameofcatz.py file uses various functions of Game class. This class has a graph, few other data structures, few private methods and some highly interactive user methods. Game() class works as the backbone of the code. Everything throughout the code, everything happens in saved in Game() documents. Users directly interact with this class only. At first glance, Game class might look just like graph class and one might say that I could have just inherited from the graph class. However there were few structural changes, so making a new class made more sense to me. Let us take example of addNode(); in graph class, a node can save any information with it. However, here letting nodes

only save ncodes seems like a more natural choice. Similarly, most methods required some level of tweaking. Therefore, I chose to go with a separate class. The Game class saves a graph in it. It has a starting node and target node set. It also uses a linked list of keep record of all paths from start to target.

DSAGraph()

This class represents a directed weighted network. It has vertices and edges. Edges connect vertices. We needed a network-type functionality in the code. Therefore, I used DSAGraph to save information about all the nodes and edges that we have. This class uses a linked list to keep track of all the vertices. This class provides various functionalities like printing adjacency lists, printing adjacency matrix and performing Depth First Search.

DSAGraphVertex()

This class represents a vertex in DSAGraph. A DSAGraphVertex has a label through which it can be identified. Vertices also have a data field to save some data. For example, we can save things like 'cost of vising the vertex'. Every Vertex also has a linked list. This linked list saves all the vertices that are directly connected with the vertex.

It was not simply possible to use just labels of vertices. We clearly needed to associate data with every vertex. Therefore, it made more sense to give graph vertices their own class. Moreover, vertices have their own behaviour. For example, updating the data saved in vertex, listing vertices that are connected to it.

DSAGraphEdge()

This class represents an edge in DSAGraph. Every edge has a vertex and a value. Edges represent connectivity between nodes. If the graph was not weighted, then there would have been no requirement of DSAGraphEdge. However, the graph is weighted, to every edge also needs to keep track of weight and therefore, it made more sense to create a different class for DSAGraphEdge.

So, We have a Game. The Game has a Graph. Graph has a linked list that can save vertices. Vertices are objects of DSAGraphVertex type. Every Vertex has a linked list. This linked list contains objects of DSAGraphEdge type. A DSAGraphEdge is made up of an object of DSAGraphVertex type and a numerical weight.

DSALinkedList()

This class provides functionality of a doubly-linked list. It is an **Abstract Data Type (ADT).** It is used at various places wherever we want to hold collection of object. I created this class as I needed to operate on collection of objects at various parts of the code. For example, collection of vertices.

DSAListNode()

This class represents the building blocks of a linked list. A block of DSALinkedList has a value, a link to previous block and a link to next block. It is important to make this class in order to make a functional linked list class.

DSAHash()

This class implements a Hash table with double hashing. The hash table stores data in numpy array by assigning a unique index to it through hash function and probing.

I created DSAHash as I needed to save Ncodes and Ecodes and it made sense to use a data structure that would allow to search for values associated with Ncodes and Ecodes in constant time O(1).

DSAHashEntry()

DSAHash is made up of these building blocks. A hash entry has a key that the hash table uses to assign it an index. It has a space to save a value and a variable to identify its state. The state can either be empty, full or deleted. It was important to create this class in order to implement DSAHash()

DSAHeap()

DSAHeap saves data in loosely formatted tree format. It is useful for sorting purposes. I created Heap because I found out that number of possible paths between two points in a graph can be significantly large. Therefore, using sorting algorithms with N^2 would make code quite slow. DSAHeap not only saves all possible path, it also sort them in O(NlogN) time complexity.

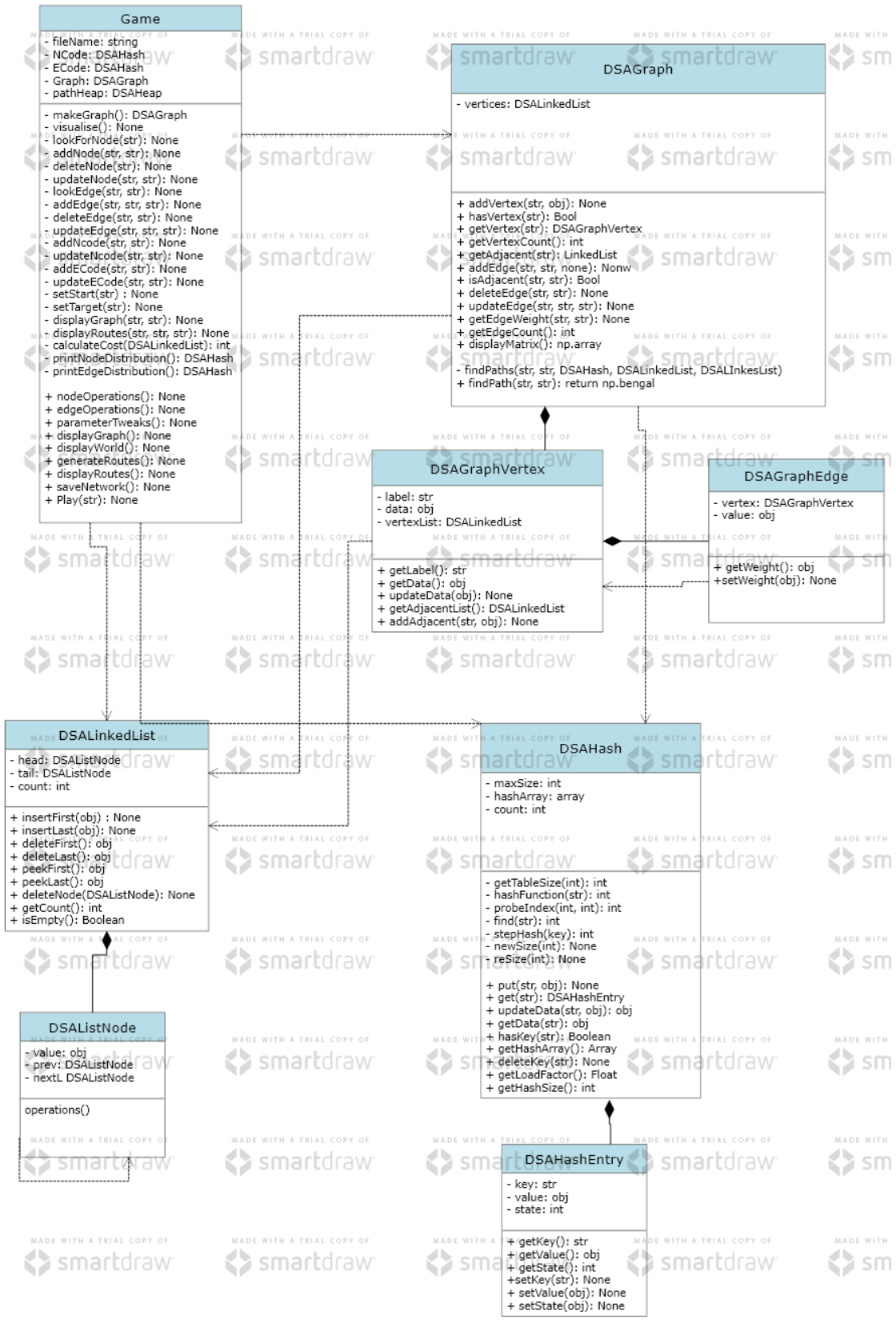
DSAHeapEntry()

DSAHeap is made up of these building blocks. A heap entry has a priority. The heap uses this priority to arrange heap entry objects. It also has space to save data.

3. Justification of Decisions

1. I am using Heapsort Algorithm to sort paths as its average time complexity is O(NlogN).
2. I am using Hash table to save Ecodes and Ncodes as it becomes easier to retrive values of them in constant time.
3. I am not extending game class from graph class as game class is more specific and its methods require modification to ensure that no wrong input gets into the network.
4. I am using a adjacency linked implementation of graph instead of adjacency matrix implementation as adjacency lists take less space and vertices can be added or deleted in constant time.
5. For the display graph option, I create a hash table to save count of vertices of each type. I use hash table here as they help in finding Ncodes in constant time.
6. I have created iterators for linked lists and hash tables. It provides functionality of running for loop over these structures.
7. I am also providing option of selecting the number of paths that a user wants to display. It is because sometimes the number of paths can be substantially large, so user should have the choice to only look at a few of the paths.
8. The paths are ranked based on ‘least energy spent.’. Weights of all the edges and nodes that are in the path is added. The path with smallest sum is displayed first. I chose this strategy as it helps in determining shortest paths.

4. UML Class Diagram



5. Traceability Matrix

Feature	Code	Text
0. Modes		
0.1 Usage	Gameofcatz.py | __printUsageInformation()	[PASSED] run program with no cmd arguments
0.2 Interactive Mode	gameofcatz.py | __interactiveTestingEnvironment()	[PASSED] run program with '-i' argument
0.3 simulation Mode	gameofcatz.py | __simulationMode()	[PASSED] run program with simulation argument, input file and output file
1. Load Data		
1.1 Load text File	Game.py | __makeGraph()	testHarness.py | test case – 54
1.2 Check for missing/invalid data	Game.py | __makeGraph()	testHarness.py | test case – 51, 52, 53
2. Node Operations		
2.1 Find Node	Game.py | _lookForNode(ndLabel)	testHarness.py | test case – 55
2.2 Insert Node	Game.py | _addNode(ndLabel, Ncode)	testHarness.py | test case – 56, 57
2.3 Delete Node	Game.py | _deleteNode(ndLabel)	testHarness.py | test case – 58, 59
2.4 Update Node	Game.py | _updateNode(ndLabel, NCode)	testHarness.py | test case – 60, 61, 62
3. Edge Operation		
3.1 Find Edge	Game.py | _lookForEdge(ndLabel1, ndLabel2)	testHarness.py | test case – 63, 64, 65
3.2 Insert Edge	Game.py | _addEdge(ndLabel1, ndLabel2, Ecode)	testHarness.py | test case – 66, 67, 68, 69
3.3 Delete Edge	Game.py | _deleteEdge(ndLabel1, ndLabel2)	testHarness.py | test case – 70, 71, 72

3.4 Update Edge	Game.py | _updateEdge(ndLabel1, ndLabel2, Ecode)	testHarness.py | test case – 73, 74, 75, 76
4. Parameter Tweaks		
4.1 add Ncode	Game.py | _addNcode(Ncode, weight)	testHarness.py | test case – 77, 78, 79
4.2 update Ncode	Game.py _updateNcode(Ncode, weight)	testHarness.py | test case – 80, 81, 82
4.3 add Ecode	Game.py | _addEcode(Ecode, weight)	testHarness.py | test case – 83, 84, 85
4.4 update Ecode	Game.py | _updateEcode(Ecode, weight)	testHarness.py | test case – 86, 87, 88
4.5 change Start	Game.py | _setStart(Label)	testHarness.py | test case – 89, 90
4.6 change Target	Game.py | _setTarget(Label)	testHarness.py | test case – 91, 92
5. Display graph		
5.1 Display (not save)	Game.py | displayGraph()	testHarness.py | test case – 93
5.2 Display (save)	Game.py | displayGraph()	[PASSED] run program with interactive mode and select the option to save when it comes to that.
6. Display Features		
6.1 print counts of features	Game.py | displayWorld()	[PASSED] run program with interactive mode and select this option
6.2 save counts of features	Game.py | displayWorld()	[PASSED] run program with interactive mode and select this option
6.3 visualize graph	Game.py | _visualise()	[PASSED] run program with interactive mode and select this option
7. Generate Routes		

7.1 Generate routes	Game.py | generateRoutes()	testHarness.py | test case – 94, 95
8. Display Routes		
8.1 display routes	Game.py | displayRoutes()	testHarness.py | test case – 94, 95
8.2 save routes	Game.py | displayRoutes()	[PASSED] run program with simulation mode
9. Save Network		
9.1 Saving network	Game.py | saveNetwork()	[PASSED] run program with interactive mode and select this option

6. Showcase

All the scenarios can be run by calling showcase.py

Scenario 1 – Running the code with a sparse graph and a dense graph and calculating range.

A complete graph is the one that has edges from all nodes to all nodes. A sparse graph would have a very low number of edges as compared to the number of edges that should be in a complete graph. 'Pacman1.txt' has a sparse graph as it has 70 nodes and only 165 edges. It means that every node has roughly 2 edges. 'gameofcatz2.txt' is a relatively dense graph with 21 nodes and 80 edges.

Area of interest – My proposition is, for a sparse graph, paths will be significantly different from each other.
How to investigate? – Create paths for a sparse graph. Take top 15 paths. Calculate range. The range should be large.
How to validate? – Create paths for a dense graph. Take top 15 paths. Calculate range. Look at the difference.

#Results – Yes, the proposition stands true. The range for top 15 routes of pacrman1.txt is 165. While for gameofcatz2.txt, it is only 3.

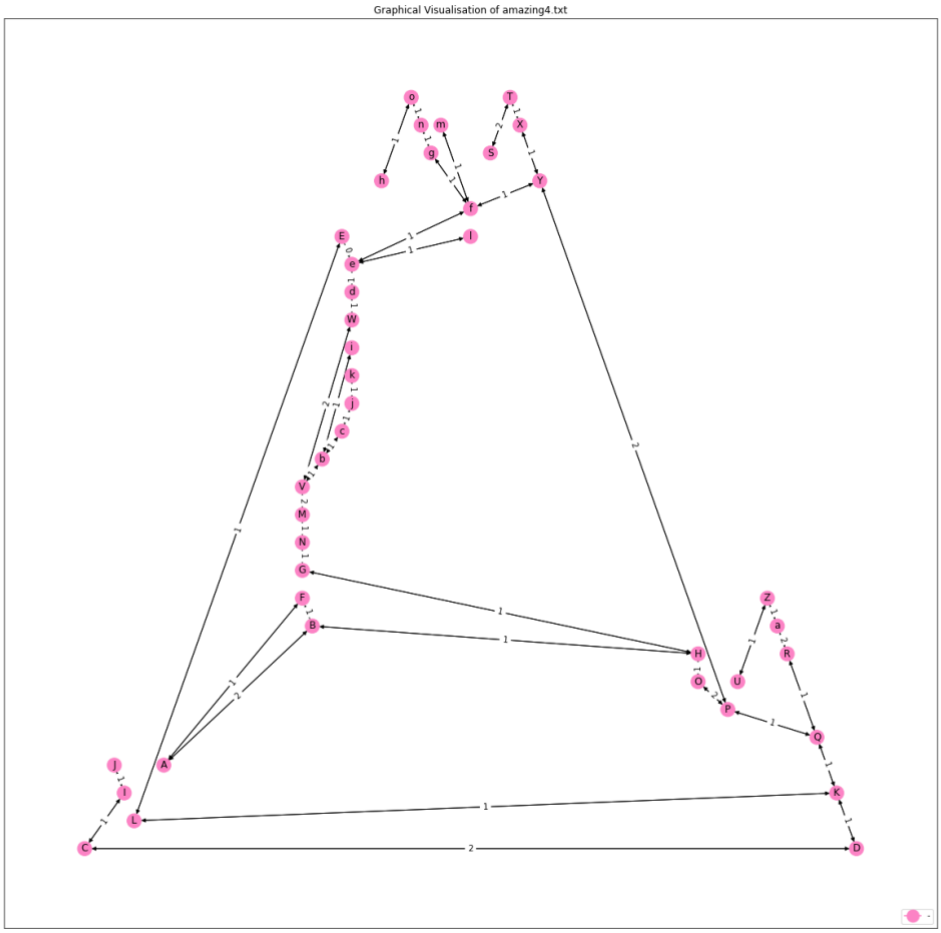
Scenario 2 – Running the code for gameofcatz.txt and finding total number of paths.

This is a plain and simple scenario. Here variable of interest is simply the number of paths. Simply set up the world as per gameofcatz.txt and calculate the number of paths possible.

#Results – The answer is 108.

Scenario 3 – Running the code for amazing4.txt and finding total number of paths and printing all the routes and sanity checking it.

The visualization for amazing4.txt is as follow –



We need to find paths from C to l.

A thorough visual investigation of the above graph reveals that there are three possible paths.

These paths are –

Weight | path

6 | CDKLEel

10 | CDKQPYfel

18 | CDKQPOHGNMVWdel

#Result – Our visual sanity check aligns with results that we get from the code.

7. Conclusion and Future Work

Graphs are non-linear structure. We can use DFS and BFS algorithms to traverse. However, these algorithms are computationally expensive and takes a lot of time if networks are large. This project tried to implement one of the networks related problem – the problem of printing all paths from source to destination in each network. It did not use any of the inbuilt datatypes of python like list, dictionary, and sets. The program can calculate all the paths and rank them based on sum of weights of nodes and edges that are in the path.

Future Works –

1. Use this code to build a GUI enabled Pacman playing bot. The code in the program can generate the best possible path at any given point of time and help the bot the act according to that.