Project Report

GameOfCatz.java

# Data Structures and Algorithms Semester 2

# User Guide

Compile the whole program using java compiler.

*:~/$ javac \*.java*

The main class is contained in gameofcatz.java.

The program has 3 separate run modes.

- Help mode: (Displays help information)

*:~/$ java gameofcatz*

Text

Description automatically generated

- Interactive mode (Opens menu and allows user to enter information directly)

*:~/$ java gameofcatz -i*

*Text

Description automatically generated*

- Simulation mode (Loads a given file; runs a simulation and then exports simulation data to the created output file)

*:~/$ java gameofcatz -s input.txt output.txt*

*Text

Description automatically generated*

**Description of Classes**

*Disclaimer\**

*Some of the classes here were originally created by Caio Marteli(me) for Practicals in Data Structures and algorithms Sem2/2021*

*They were further adapted and improved upon for utilisation on this project see the header comment on the individual classes for further information.*

*Marteli, C (2021) DSAPrac4 source code (Version 2.0) [Source code]. https://github.com/cMarteli/DSA*

**gameofcatz.java**

The starting point and main class of the project. Its purpose is to handle the provided user command line parameters and choose which operation to boot into, it also serves as a last line of error handling by being the only place where a general exception is caught (all others try to get catch a specific exception usually thrown on purpose by my classes.)

This class allows the user to boot the program up into three modes, the first shows instructions on how to correctly write the command line parameters. The second opens a command line menu interface which provides access to all methods related to the graph. Thirdly, we have simulation mode which loads a file placed in the same directory as the program and provides the user the ability to choose a save file which will hold output data related to the simulation.

**UserInterface.java**

The menu system of the whole program. It has a main menu as well as sub menu methods for the Edge and Vertex operations. This class also contains methods for handling user input such as checking for integers and strings the reason it was designed this way was to avoid code repetition and improve readability, as the user is asked to input commands several times.

The main menu handles method calls to the appropriate method to handle each function of our graph.

[1]Load input file

Allows user to load an input file of their choosing.

[2]Node operations

Opens the node operations submenu. This is another method contained in this class.

"[1]Find\n[2]Insert\n[3]Delete\n[4]Update\n[5]Back to menu"

[3]Edge operations

"[1]Find\n[2]Add\n[3]Remove\n[4]Update\n[5]Back to menu"

[4]Parameter Tweaks

[5]Display Graph

Displays the vertices currently in graph as a list in no particular order.

[6]Display World

Traces a route through a given graph given a start point.

[7]Generate routes

[8]Display routes

[9]Save network

[0]Exit

**FileReader.java**

This class is responsible for file input and output handling. This was the only class chosen to be static. As you would never need to instantiate more than one file reader at a time and probably would not want to due to memory concerns for eg having several scanners open at the same time.

This allows for further improvement in readability and memory as we do not need added variables being made every time a file needs to saved or read.

**DSALinkedList.java**

Contains all information methods pertaining to the linked list data structure created for this unit. It is a double ended doubly linked list, meaning it store the head and tail information as well as next and previous on the node class.

This was chosen to allow for a reliable and easy to implement version of remove at and search, which helped greatly when implementing the graph function of remove vertex and remove edges which themselves are required function in this project outline.

Having a tail reference in the list also allows for an insert last without having to iterate through the list every time decreasing time complexity that in turn allows for a straight forward implementation of graph methods such as addEdge:

        public void addEdge(DSAGraphVertex inVertex)

        {

            links.insertLast(inVertex); //inserts at the end of links

        }

The improved readability as well as decrease in computational time justify the increase memory required for this type of list.

**DSAListNode**

A private inner class of DSALinkedList, this design choice was made for access modifier safety. I do not want methods outside of the list itself to be able to modify its values and by making it an inner class I’m able to restrict how that memory is manipulated to only the accessors and mutators of the parent class itself. It contains an iterator and implements serializable. The iterator allows for an easy way to cycle through the linked list and the serializable interface allows for ease of saving files.

PS. These classes were originally designed by me to utilise JAVA generics, but I encountered problems when trying to implement serialization alongside generics as serializable is already a generic class.

Given more time and if implemented correctly this would be an improvement over the current version of the program as generics allow for much stronger type-safety as they are cast immediately at instantiation.

**DSAGraph.java**

Arguably the most important and complex data structure of the project. This graph uses an adjacency list and not a matrix.

This was chosen due its ability to quickly iterate over all edges because any node neighbour can be accessed directly as neighbour vertices are always easily accessible as a list whereas in a matrix an entire row must be scanned adding to time complexity.

It also is quicker at adding and deleting nodes since it does not need to keep track of a 2D array and its index values. It is also fast adding edges however so is matrix.

The downside it is not as efficient for a 1 to 1 graphical representation as you could easily parse the matrix to a drawing function which in turn could be used to display all vertices in a grid. Had that been a requirement this would have been given more importance.

**DSAGraphVertex**

**DSAQueue.java**

**DSAStack.java**

**UnitTestHarness.java**