Project Report

GameOfCatz.java

# Data Structures and Algorithms Semester 2

# User Guide

This simulation allows you to traverse through a ZOO as a cat. Each node represents an area a different animal resides in. The cat wants to traverse to all areas (to eat the other animal’s lunches) it may only do so if the cages are connected to each other.

The user may run any number of simulations specifying the start point they may also set an end point for the cat to always end on. If nothing is set the cat will stop moving after travelling to all areas.

How to use:

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*

[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.

Node operations

Current vertex count: //show how many vertices are in graph

>>Find //Checks if a node currently exists given a label

>>Insert //Inserts a new node

>>Delete //Deletes a node

>>Update //Updates Node information given it exists

[3] Edge operations

>>Find //Finds an edge given two nodes

>>Add //Inserts a new edge given two nodes

>>Remove //Removes a new edge given two nodes

[4] Parameter Tweaks

Allows user to add a fixed end point for every simulation.

[5] Display Graph

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

[6] Display World

Displays all nodes all links in between.

[7] Generate routes

Asks the user to input the number of simulations and run that many will ask for input for a start point in every simulation

[8] Display routes

Displays all routes currently in the list queue

[9] Save network

Saves the current graph as a SERIALIAZED FILE

[0] Exit

Exits Program

Simulation mode

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

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

*Text

Description automatically generated*

NOTE: *Do not type in file extensions as the program automatically puts them in. Serializable files are save as “.out” and output files are saved as “.txt” text input files are also assumed to be “.txt”*

The output file is the traversal of all simulations. It is human readable only. To save a created Graph please use Interactive mode and select option [9] from the main menu.

It will then ask the user to select the number of simulations to run and will print to a file all the simulations it ran.

# Input file and showcase

This program uses custom input files

Comment characters are ignored

Files must begin and end with a # character

Only NODE and EDGE are permitted as commands

Eg.

#

Node Elephant

Node Rabbit

Node Dog

Node Ant

Node Mongoose

Node Narwhal

Node Snake

Node Cat

Node Bear

Edge Elephant Rabbit

Edge Elephant Bear

Edge Rabbit Bear

Edge Rabbit Dog

Edge Rabbit Snake

Edge Rabbit Cat

Edge Dog Ant

Edge Dog Narwhal

Edge Snake Rabbit

Edge Snake Cat

#

Visualization:

**A picture containing indoor

Description automatically generated**

**Description of Classes**

\*\*\*Disclaimer\*\*\*

Some of the classes in this project 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. This class handles all manners of direct user input. 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.

**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 example having several scanners or file writers 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 save 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 simple 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**

A private inner class of DSAGraph.java, again this is made to only allow methods from inside the parent class to be able to modify class fields.

**DSAQueue.java**

This class is used to store the list of results after every traversal of the graph. A queue was used due to the need to these values to always be added in, in the order they occurred a linked list could’ve been chosen but using a queue restricts the methods which could be used to add to the queue such as insertAt.

**DSAStack.java**

This class is used during Depth First Search algorithm implementation.

visited.push(vx); //push onto visited stack

                Iterator<DSAGraphVertex> itr = vx.getAdjacent().iterator();

                do{

                    while (itr.hasNext())

                    {

                        vx = itr.next();

                        if(!vx.getVisited()) //if not visited traverse here

                        {

                            queue.enqueue(vx); //adds to output queue

                            vx.setVisited(); //sets to visited

                            dfs(vx, visited, queue);

                        }

                    }

                    visited.pop();

                } while(!visited.isEmpty());

Stack behaviour is needed here as every time we visit the links of a node we store it’s parent node in the stack and go through all of its links until they are all marked as visited then we pop from the stack until we are done.

**UnitTestHarness.java and GraphTestHarness**

UnitTestHarness is used for testing all of the methods of data structures except for the graph which has its own test harness. I chose to do it this way as it was getting a bit too crowded having all in the same test harness.

Conclusion

The program as it stands is missing some core features such as having the modifier attributes to each note and in turn having those modifiers affect traversal. For example, having an area with a dangerous animal that can only be approached from a certain side or maybe not at all. Or giving the cat the ability to pick up items which give it stat boosts.

Another improvement would have been the use of Java generics for improved type safety over Objects.

Having weight and direction on the edges to represent distances and one-way gates and manhole covers as shortcuts which could be represented as edges that can only be traversed in specific conditions.