Preliminary Project Report

*By Marcellino Samuels*

# Introduction

This idea for this project originated from the desire to find an optimal route within a video game. The game has a series of nodes that can be connected together to create a network. Within this network the play has the opportunity to gather resources at each node. However, each node has a cost and players are given a finite amount of points to create their network. This type of problem fits perfectly into graph theory as the network translates directly to a graph. There are a large number of algorithms that could then be used to find an optimal path or network.

A natural algorithm has been selected due to its ability to utilise randomness, which should lead to a variety of solutions in addition to an optimal solution. This variation is beneficial as this gives rise to the potential of multiple optimal solutions being found. In contrast a more systematic algorithm would often result in one optimal solution regardless of how many other valid solutions exist within the graph.

As this project aims to find a minimum tree of a graph it can be applied to any problem that can be represented as a graph and solved by finding the minimum spanning tree. This means that there are many applications for this in the field of computer science (Riaz, F. and Ali, K. 2011). However, this range of problems also falls within the scope of graph theory. Graph theory is a field of mathematics which uses edges and nodes/vertices to represent relationships between objects. The origins of graph theory can be traced back to the Seven Bridges of Konigsberg problem 1735 (Newman, 2000). This was solved by Leonhard Euler a well-known mathematician of his time. The method he created to overcome this problem became the foundation of graph theory (Wilson, James & Lloyd, 1976).

An example of an application is the following:

Each node represents a location and the value of each node represents its importance in the trade network. The weights of each edge represent the distance between each location. The algorithm will try to find the highest value network which minimises the distance travel.

# Aims and Objectives

This project aims to solve the following problem using an implementation of Stochastic Diffusion Search (SDS) algorithm. Given an edge weighted graph where each node has been given a value, find the optimal minimum tree which visits X nodes and maximises the values of the nodes within the tree, where X is a number of nodes in the tree.

Theoretically the SDS algorithm should be capable of producing optimal spanning trees for any given graph that contains a minimum spanning tree. This project aims to prove that SDS is capable of finding optimal solutions by comparing the results against Krushkal’s algorithm.

The project also aims to measure the exploitation and exploration of SDS on edge weighted graphs. If necessary a set of benchmarking graph will be create with the intention of finding the limitations for this design of SDS.

While it is important to optimise a program this project is mostly interested in the testing and analysis of SDS for edge weighted graphs. In this sense the code is a more of a tool to allow for the collection of data.

# Methodology

For this project a more flexible waterfall model known as the incremental build model is in use. This model design implements and tests the program incrementally. Following this development model allow for the program to be built and tested one class at a time. This incremental style of development is perfect for incorporating Test Driven Development (TDD) as this has short development cycles that ensure that the program is create one class at a time, thus helping prevent large errors which persist through multiple classe. It also helps break down the project into smaller more manageable parts much like a divide and conquer strategy.

The development cycle for each new class is the following:

* Create requirements that the code my adhere to
* Design the code based off the requirements
* Implement Unit tests to insure the requirements are meet
* Check that the test fail on the skeleton implementation of the class
* Implement the code to pass each test one at a time
* Once all tests have been passed the code is uploaded to Git

# Project Plan

The current project plan is still undergoing development as research is being conducted to find the correct method of creating a Gantt chart. This is the current project plan, there are still more milestone that need to be created and added.

## Week 1:

* Create Git Repository
* Select a development method
* Select and IDE and programming language for implementation

## Week 2

* Research testing methods and select an appropriate one
* Create the final report structure and sections
* Estimate a rough word count for each section of the report

## Week 3

* Create requirements for the Edge Weighted Graph (EWG) and subsequent classes
* Implement unit tests for Edge class
* Pass all test for Edge class
* Update report

## Week 4

* Implement unit tests for Vertex class
* Pass all test for Vertex class
* Update report

## Week 5

* Implement unit tests for EWG class
* Pass all test for EWG class
* Update report

## Week 6

* Create requirements for stochastic diffusion search (SDS)
* Implement unit tests for SDS
* Update report

# Week 7

* Create a simple graph and use Krushkal’s algorithm to find the optimal solution
* Run SDS and record the results
* Used results to show if SDS returns optimal solutions
* Test the exploration and exploitation on given graphs

## Week 8

* Create benchmarking graphs
* Test the exploration and exploitation on the set of graphs
* Analyse the results and update the report

# Implementation (Progress to date)

The java implementation has been constructed using object-oriented programming (OOP). The variables for the EWG, Edge and Vertex classes are private with setter and getters to incorporate encapsulation.

The first stage of the implementation was to create an edge weighted graph as a framework for the program. An edge weighted graph consists of two main components which are represented in the java implementation by an Edge class and a Vertex class. These classes create the foundation for the Edge Weighted Graph class that can theoretically convey any graph. In order to traverse the edge weighted graph both classes need to retain information which pertains to the other class.

A vertex or node is a point within the graph that is often labelled. These can be connected to each using edges. These ideas translate into the following variable:

* String label – Allow for the vertex to be labelled
* HashSet<Edge> edges - A set of all edges that are attached to this vertex

Each new Vertex class is instantiated with a String variable as an argument. The constructor uses the String as the label for the class then creates a new HashSet<Edge>. A string has been used for the label to allow any character to denote the vertex. The HashSet<Edge> has been implemented as it is a dynamic data structure that will prevent duplicates of each edge classes in the set. An addition benefit of this data structure it that the search time for an Edge will be constant. This will help reduce the time taken for each traversal which is important as the method that will be used to check for loops when generating a new hypothesis and within Kruskal’s algorithm is the depth first search traversal.

For the vertex class the getter and setter are used to manipulate the HashSet<Edge> and to update and access the String variable. The methods that interact with the HashSet<Edge> are named add() and remove(). The add method takes an edge as an argument, then checks if an existing edge has the same connections. If an existing edge has the same connections it will not be added to prevent duplicate edges from occurring. Otherwise, the edge is added to the HashSet<Edge> thus connecting the vertex class to another vertex class within the graph.

An edge is used to connect to vertices within a graph. For an edge weighted graph the edge will have a weight linked to itself. These concepts translate into the following variables:

* Vertex Vertex1 – A vertex class which the edge is connected to
* Vertex Vertex2 – The other vertex class which the edge is connected to
* double Weight – A double that represents the weight attached to the edge

An array has been used to store the two Vertex classes as the maximum number of vertex is predetermined to be two.

This information is passed to the class each time a new Edge class is instantiated.

The class constructor has been overloaded to allow for two constructors. This allows for the representation of regular graphs and for weighted graphs. As such the first constructor takes two Vertex classes and sets the weight to 0 to express a regular graph. The second constructor takes two Vertex classes and a numerical value (double) for the weight to express the weighted graph.

A private method named initialize() has been created which takes two vertex classes as an argument. The method calls the add() method of each vertex class given to add the newly created edge (this) to the edge sets of each vertex class. This method has been created private to enforce encapsulation as the initialize() method is only required within the edge class.

The Edge class has been given a special getter method to help with the traversal of the graph. The method is called getOther() and takes a vertex as an argument. The method will then check if the vertex is connected to the node. If the vertex is connected to the node, then method will return the vertex that is connected to the over side of the edge. If the vertex is not connected to the node then with method returns null.

# Testing and Errors (Progress to date)

Requirements for Edge Weighed Graph

* The ability to construct regular and edge weighted graphs
* The graph should not allow vertices to be connected to themselves (no self-loops)
* There must be no duplicate connections between vertices
* The label of each vertex must be unique

Requirements for Edges:

* Must be connected on both sides as a vertex cannot be connected to nothing
* The graph will not contain any vertices that self-loop
* The edge must be able to return both of the vertices connected for traversal

Requirements for Vertex:

* The ability to add and remove edges to allow for construction of graphs
* Each vertex must know how many edges are connected to it at any given time
* Each vertex must be able to return specific edges for traversal

An error occurred when testing the prevention of duplicate edges within the vertex class. It was caused by vector.equals(otherVector) not recognising when two vectors are equal. This was due to the creation of a new vertex within the test. The solution was to check the labels (String) variables of each vector against each other. This has overall made the program more robust as it will now be able to recognise if newly created vertex and edge classes carry the same information as an old class.

# Planned Work

Updating the project plan to a Gantt chart is the top priority for the project in its current state.

The Edge Weighted Graph (EWG) skeleton has been created along with the requirements for the class. These requirements will be implemented then the EWG class will be completed thus finishing the foundation for the project. Upon completion, the final project report will be updated to include details of the implementation, testing and errors for all classes up to this point.

The next step will be to create requirements for the stochastic diffusion search (SDS) algorithm and implement the tests in Java. As the java implementation of the algorithm has been designed the creation of this class should take less than one week to complete.

Once the implementation has been complete tests will be carried out in order to measure the exploitation and exploration of SDS on the edge weighted graph. These tests will be carried out on a set of benchmarking graphs that will become increasingly complex. This will assess the capabilities of SDS for edge weighted graphs and hopefully find limitations as well as potential areas for improvement.

Finally the program will be modified to find trees of a specific size as well as the minimum spanning tree. The vertex class will be updated to contain values and a new fitness function will be added to balance the importance of shortest path versus the value of each vertex.

# Bibliography

Newman, J. (2000). *The world of mathematics.* Mineola, N.Y.: Dover Publications.

Riaz, F. and Ali, K. (2011). Applications of Graph Theory in Computer Science. *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks.*

Wilson, R., James, W. and Lloyd, K. (1976). *Graph theory, 1736-1936.*