# Abstract

# Acknowledgments

# Table of content

# Introduction

# 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 value of the tree while maximising the value of the nodes, where X is a number of nodes in the tree.

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.

The project implementation of SDS will be in java using the eclipse IDE which is free and readily available online.

# Applications

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 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 would be 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.

# Algorithms

# Implementation

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.

For the following example the vertex class variables will be named V1 and V2 respectively:

## (Code)

# Testing and Errors

Test Driven Development (TDD)

Requirements for Edge Weighed Graph

* 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
* Vertices can exist without any connecting edges
* Vertices can exist with connecting edges
* Traversal between all connected vertices is possible
* Edges cannot exist without connected vertices
* The graph can be deconstructed

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

Requirements for SDS Initialization phase:

* Agents are generated during the initialization phase
* Every agent has a hypothesis after the initialization phase

Requirements for SDS Test phase:

* Every Agent’s fitness is calculated
* The total fitness of all agents is equal to 100
* Agents within the activation threshold become active
* Agents outside of the activation threshold are inactive

Requirements for SDS Diffusion phase:

* Every agent checks a random agent’s hypothesis
* Agents copies hypothesis if active agent selected
* Agents generate random hypothesis if inactive agents selected

Requirements for Agents

* Agents are set to inactive by default
* Able to generate random hypothesis
* Hypotheses are spanning trees
* Hypothesis must be accessible for external classes

Requirements for DFS

* Able to identify cycles in graph
* Able to identify spanning trees
* Traversal is Depth First Search

Error when preventing duplicate edges caused by vector.equals(otherVector) not recognising when the vectors are equal. This was due to the creation of a new vertex within the test. The fix 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. (Vertex Tests)

Error with test for random hypothesis, was checking class EWG against class EWG which resulted in different EWG which had the same nodes and edges. Changed test to be based on the weight of the graphs. (Agent test)

Error when checking if a single node graph with no edges is a spanning tree, the result returned was true when it is supposed to be false. Revised the if statement that set spanning tree check to true and added a condition for single node graphs as they will not be spanning trees or contain cycles due to the graph not allowing for self-looping vertices.

# Methodology

Waterfall

# Research

# Results

# Future Development

# Conclusion

# 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.*

# Appendix