# Abstract

# Acknowledgments

# Table of content

# Introduction

Description of MST problem

Description of DC-MST

Aims and Objectives

Applications

Road map of content

# 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

SDS

Quick sort

DFS

Krushkal’s

Stochastic Krushkal’s

# 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)

The EWG class is used to combine both the Edge and Vertex class together to simplify the construction of graphs. This allows for the implementation of external logic which constrains the graph to be an undirected graph.

A EWG contains vertices and edges, both of these concepts have been expressed as separate classes. The EWG uses two data structures to hold the information of all the Edge and Vertex classes:

* HashSet<Edge> edges
* TreeMap<String, Vertex> vertices

The HashSet data structure has been selected to store Edges as it has a constant time complexity for adding and searching stored information. This is useful for the generation of random hypothesis as all edges except one will need to be accessed to create the MST.

A TreeMap has been used to store all Vertex classes used for the graph using a String as the key to allow for Vertices to be stored by a simple label. This data structure automatically arranges the Vertices by their labels which creates a logical order for the toString() function. The TreeMap prevents duplicate Vertex labels thus ensuring that each vertex has a unique label.

The EWG class has been overloaded to allow for three constructors. The first constructor is a blank constructor that takes no arguments then instantiates the HashSet<Edge> and the TreeMap<String, Vertex> to allow for graph to be constructed from an empty graph with no vertices or edges. The second constructor takes a TreeMap as an argument which is used to input vertices. This allows for a graph to be created with all vertices present from the instantiation of the class. This is very useful for generating random graphs as the edges can be added separately in a similar style to Krushkal’s algorithm. The final constructor takes an EWG as an argument and creates a clone of both the HashSet<Edge> and TreeMap<String, Vertex> to allow for the alteration of the EWG given in the argument without effecting the original EWG. This constructor is useful for the generation of randomised graphs as the resulting EWG varies from the original.

The EWG class has been given a large variety of method for the creation and deconstruction of graphs. In addition to this methods for extracting information about the graph or other features within the graph, have been produced to support encapsulation within the class.

Two important characteristics of a graph are the number of vertices (order) and the number of edges (size). These characteristics are directly linked to the supporting data structures allowing for the order() and size() method to be extrapolated from the size of the corresponding data structure. The order of the graph is the size of the TreeMap and the size of the graph is the size of the HashSet.

In order for traversal to be possible the Vertex classes stored within the EWG must be accessible externally from the class. As the Vertex classes are stored within a TreeMap, to retrieve a specific Vertex the TreeMap must be searched. This is achieved by the getVertex(String label) class which takes a String as an argument. The String is the label of the Vertex that is desired to be returned. If the String is found the method returns the appropriate Vertex, if the String cannot be found then the method will return null.

During the testing phase of SDS the total weight of the all the edges within the EWG is required. The method weight() has been created to fulfil this purpose. This method iterates through the HashSet<Edge> to sum the total weight of all edges and return the result.

The method called addVertex() has been overloaded to allow for two different variable types. The first type is a String argument, when used the method creates a new Vertex class using the given String as an argument. The new Vertex class is then added to the TreeMap using the String argument as the key. This allows for the creation of new Vertex classes directly through the EWG class. The second type is a Vertex class argument, when used the method adds the Vertex to the TreeMap using the String label variable of the Vertex class as the key for the TreeMap. This allows for pre-existing Vertex classes to be added to the EWG class.

There is a large variety of way in which an edge can be added using this implementation of the EWG. As a result the addEdge() method has been overloaded numerous times. One addEdge(Edge e) method takes an Edge as an argument to allow for the addition of pre-existing Edges. However, to comply with the logic of an Edge both Vertex classes connected to the Edge class must be checked to see if they exist within the EWG. If the Vertex class does exist the edge is re-connected to the duplicate vertex that exists within the EWG. This is to prevent duplicated of the same Vertex class from appearing within the EWG. If the Vertex class does not exist it is added to the EWG as an Edge cannot point to nothing.

The remaining addEdge() methods are similar as they create the Edge within the EWG class itself. There are four more variations for each of the addEdge() methods the first being addEdge(String vLabel1, String vLabel2, double w) which takes two Strings and one double. Each String is passed to getVertex(String) to find the corresponding Vertex class. A Edge class is then created to connected the two Vertex classes using the double as the weight of the edge. The second variation of this method is used when creating unweighted graphs and uses only two Strings as the arguments. This method calls the original as the only difference is that the weight is set to 0. For the third variation of the addEdge(Vertex v1, Vertex c2, double w) the Strings are replaced with Vertex classes. Once again a new Edge class is created using the arguments passed to the method. The final variation takes two Vertex classes and calls the third variation with 0 as the double allowing for the creation of unweighted graphs.

Overloading the addVertex() and addEdge() methods has allowed flexibility when constructing new graphs as there are multitude of ways in which the vertices and edges can be added to the graph. They can be create both internally and externally from the graph and there is no strict order in which the vertices or edges must be added. This freedom helped to ease the assembly of the tests for the EWG class.

To facilitate the deconstruction of an EWG the removeEdge() and removeVertex() methods have been created. Both methods have been overloaded to incorporate a variety of possible arguments that can be used to reference Vertex or Edge classes. This is to keep the conversion of String to Vertex from occurring externally as this could cause inconsistencies within the code and lead to error.

The removeVertex() method has been overloaded to allow for two different variable types, String and Vertex class. Both removeVertex() methods have the same goal, which is to remove the Vertex corresponding to the given argument from the EWG. As a result the removeVertex(String vLabel) passes the String vLabel to the getVertex(String) method to locate the appropriate Vertex. This Vertex is then passed to the removeVertex(Vertex v) method. The removeVertex(Vertex v) method extracts the HashSet<Edge> from the Vertex v to iterates through the HashSet<Edge> and remove all edges connected to the Vertex v. This it to prevent the edges attached from pointing to null after the Vertex v is removed. Once this has been completed the Vertex is removed from the TreeMap and deleted.

To incorporate the removal of Edge the removeEdge() method has been implemented. This method has been overloaded with three different constructors. The main removeEdge(Edge e) method takes an Edge e as an argument then removes Edge e from both of the Vertex classes connected by Edge e, after the Edge e is removed from the HashSet and deleted. The next constructors take two String then uses getVertex() to convert them to the appropriate Vertex classes. From this point the method is identical to the final constructor which takes two Vertex classes as an argument. Using both Vertex classes the findEdge() Vertex method can be used to find the interconnecting Edge class. This Edge class is passed to the main removeEdge(Edge e) to complete the process.

During the implementation of the Stochastic Krushkal’s algorithm is became apparent that the EWG would need have all Edge removed from the graph while keeping the information of which Vertex classes each Edge class connects. To accomplish the removal aspect of this idea the removeAllEdges() method was designed. This method iterates through each Vertex class and to clear their HashSet<Edge>. Once this has been completed the EWG HashSet<Edge> is cleared as well leading to a graph of only Vertex classes.

Agents are a fundamental part of SDS, as the communication between the each agent allows them to gradually find an optimal solution. Agent are required to create hypothesis and decide whether they are active or inactive. These ideas translate into the following variables:

* Boolean status
* EWG hypo
* double fitness

Constructor talk

A Boolean has been used to determine whether an agent is active or inactive. This is due to the binary nature of an agents status. If the status Boolean is “false” the agent is inactive, if the status Boolean is “true” the agent has become active.

Next is the agents hypo (an abbreviation of hypothesis) which is a randomised spanning tree of the EWG given as an argument when the class is instantiated. This is stored as an EWG as a spanning tree is a sub graph of the original EWG.

Gen Hypothesis talk

SDS

Why these data structures are chosen

How they represent the graph

General outline of methods

The EWG class contains various ways to added and remove edges and vertices (expand on this)

Agent Class – DFS – SDS

# Testing and Errors

Test Driven Development (TDD)

What TDD is and why I use it.

List of requirements

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
* Every agent’s fitness is calculated

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

Why these tests are chosen. How they are implemented. Errors that occurred during the process.

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