# Abstract

# Acknowledgments

# Table of content

# Introduction

This project originated from the desire to create an optimal network of nodes within a massive multi-player online game. Each node has been assigned a value that represents the cost to connect the node to the player’s network. Players are given a finite number of points to create their network. Nodes fall into 3 categories:

1. Resource nodes - contain recourses that can be gathered by workers.
2. Worker nodes - contain workers which can travel along the paths to work on recourse nodes.
3. Neutral nodes - contain neither workers nor resources.

The time taken *T* for a worker to complete its task is proportional to the distance travelled *D* and the work speed *S* of the worker. Let *T* = *D* \* (1 – *S*) where *D* **ℝ+** is a positive real number and *S* **ℝ+** {0,100} is a positive real number between 0 and 100 exclusive. The goal is to minimise the overall time taken by each worker and to minimise the total cost of the player’s network.

To simplify this to a minimum spanning tree (MST) problem it will be assumed that the player will have enough points to create a network containing all nodes. Now the minimisation of the overall time taken by workers fits perfectly into graph theory as the network can be translated into an undirected weighted graph as each node is a vertex (node) and each path is an edge. The weight of each edge is the distance from one node to another.

The MST problem (IN REF) is one of the most common and well-known optimisation problems in the field of computer science. This is mostly due to its close relationship with graph theory [[1]](#footnote-1) which is a field of mathematics that uses edges and nodes to represent relationships between objects. As a result real world problems such as communications, power, transportation and many more can be unravelled by solving the MST problem. During the inception of the MST problem various independent sources and algorithmic solutions where created (REF). The first efficient solution to be published was developed by Boruvka in 1926 (REF), despite this the two that became the most popular are known as Kruskal’s (REF) and Prims (REF) algorithm.

However these algorithms are deterministic which mean that only one solution will be returned even if there are multiple optima. As the MST problem is a sub problem of the worker’s time optimisation (IN REF), it will be necessary to view all optimal as well as some sub-optimal solutions as potential solutions due to the fact that they may yield better times as the distances from worker nodes to resource nodes is an important factor. As a result of this the focus of this project will be to discover whether Stochastic Diffusion Search (SDS) is able to create MST from various benchmarking graphs. A natural computing algorithm has been selected for its ability to utilise stochasticity, which should lead to a variety of solutions in addition to an optimal solution. This variation is beneficial as this may gives rise to the potential of multiple optimal solutions being found.

This project aims to implement multiple variations of the SDS algorithm to run tests that will be carried out on a set of benchmarking graphs that will become increasingly more complex via the size (IN REF) and order (IN REF) of the graph. The data produced from these tests will be used to compare and contrast the variations of this algorithm. This will assess the capabilities of SDS for MST using random graph generation and hopefully find the limitations as well as potential areas for improvement. It will be interesting to see the results and how this algorithm compares to other algorithms such as Kruskal’s (REF).

Altering the activation techniques for the agents should affect the convergence of the algorithm. It will be interesting to see the variations between each algorithm and examine the subtle difference that will occur as a result of these changes.

ROADMAP OF CONTENT -------------------------------------------------------------------------------------------

# Literature Review

Let *G* = (*V*, *E*) be a connected weighted undirected graph, where *V* = {*v*1, *v*2,…, *v*n} is the a finite set of vertices/nodes and *E* = {*e*1, *e*2,…, *e*n} is a finite set of edges that create connections between vertices in set *V*. Every edge has a weight *W* = **ℝ+** that is a positive real number.

Degree

Order

Description of MST problem

# Algorithms – (ADD FIGURE EXAMPLES FOR ALL)

Stochastic Diffusion Search

Stochastic diffusion search (SDS) is a population based optimisation algorithm created by J.M. Bishop in 1989 (REF). J. M. Bishop developed SDS with the intention of using stochasticity to find efficient solutions for various problems (REF). This idea is what leads to the creation of SDS which by extension created the swarm intelligence metaheuristic (REF) as SDS was the first algorithm to utilise this as a tool for optimisation.

The SDS algorithm is comprised of 3 phases:

1. Initialisation - In this phase agents generate a random hypothesis (solution) to the problem.
2. Test - Each agent will evaluate their hypothesis to determine which category they will join. If an agent’s hypothesis is valid they will become active. Otherwise, the agent will become inactive.
3. Diffusion - Each inactive agent *L* will communicate with a random agent *R*. If agent *R* is active, then agent *L* will copy agent *R*’s hypothesis. If agent *R* is inactive the agent *L* will generate a new random hypothesis.
4. Repeat from step 2 until a termination condition has been met. A common termination condition is to terminate after a set number of iterations.

Stochastic Diffusion Search for minimum spanning trees

In order to apply SDS to as a solution to the MST problem, features of the algorithm have been modified while leaving the core ideology of the algorithm the same. Three different variations of SDS have been created in an attempt to solve the MST problem. For all versions of SDS the initialisation and diffusion phase are the same, the differences lie in how the fitness functions are calculated and the technique used to activate agents within the Test phase. Another change that has been included is the addition of an update phase which takes place during the diffusion phase. This phase can be seen as a mutation of an agent’s hypothesis.

The following section will cover each variation using the labels SDS 1, SDS 2 and SDS 3 for each respective variation of the algorithms.

The initialisation phase follows the same process as the original SDS by generating the population of agents. The first alteration is the agent’s hypothesis to the problem. For all versions of SDS for MST the agent’s hypothesis will be a randomly generated spanning tree. This will be generated using a newly developed algorithm called Stochastic Kruskal (IN REF).

The test phase is a crucial phase for all versions of SDS as this is where the algorithms drastically change to become unique versions of the SDS algorithm. The technique developed for selecting active agents for SDS 1 has been inspired by roulette selection from the genetic algorithm.

The test phase for SDS 1 is defined as the following:

1. For each agent sum the total weight of the agent’s hypothesis *H* = *w* (*H*). let function *w* be defined as follows: (EQ)
2. A temporary fitness *T* is calculated for each agent by taking the sum of the total weight for the original graph *G* = *w* (*G*) then subtracting *H*. The larger the fitness the better the agent’s hypothesis.

*G* - *H* = *T*

1. Each agent’s temporary fitness *T* is divided by the sum of all agents' fitness *S* to give a percentage. This percentage is the overall performance of the agent in comparison to all of the agents and will be used as the agent’s fitness *F*.

*T* / *S* = *F*

1. The agents are then arranged in descending order of *F* and the top *X* become active, where *X* is a percentage between 1 and 100, this will be known as the activation rate for the algorithm. This percentage is a parameter that can be adjusted that which will affect the convergence of the algorithm.

The testing phase for SDS 2 uses the roulette select method to decide which agents become active. The first 3 steps of SDS 1 are based off roulette selection which enables them to be re-used as part of SDS 2. This can be seen as an extension of SDS 1, as the value *X* is once again used as a deciding factor of the agent’s activation rate. For an elitist approach the agent with the best fitness is set to active before continuing the process as normal.

The next steps are a continuation from step 3 of SDS 1. The test phase of SDS 2 is defined as the following:

1. Create a roulette wheel based off the fitness of each agent such that the larger the value of *F* the larger the section on the roulette wheel. (FIGURE)
2. Generate a random number between 0 and 100 exclusive and the selected agent which this value corresponds to will become active. Active agents are removed from the roulette wheel.
3. Continue step 2 until the sum of all active agents’ fitness is larger than *X*.

The final variation of SDS for spanning trees uses a more traditional and widely used approach for the activation of agents. This approach is very similar to the diffusion phase of SDS at its core.

The test phase for SDS 3 is defined as the following:

1. For agent *A* select another random agent *R* where *A ≠ R*.
2. If agent *A* fitness is better than (less than) agent *R*’s fitness then agent *A* becomes active.
3. Repeat steps 1 and 2 until every agent has communicated with another agent.

Last is the diffusion phase, this take place after the test phase of each SDS variation and is the same process for each of them. During this phase each inactive agent *L* will communicate with a random agent *R*. If agent *R* is active, then agent *L* will copy agent *R*’s hypothesis, once copied agent *L*’s hypothesis will be updated. If agent *R* is inactive the agent *L* will generate a new random hypothesis.

The Update phase is defined as the following:

1. Compare the agent’s hypothesis *H* to original graph *G* to create a new set of edges *U* that is defined as *U* = { *G* | *H* }.
2. If *U* is empty end algorithm.
3. Randomly select an edge *E* from set *U* then remove *E* from *U*.
4. Add *E* to hypothesis *H* then check if *H* has a loop.
5. If there is a loop in *H* create a new set *C* of all edges within the loop. If there is no loop remove *E* from *H* then repeat step 2 to 5*.*
6. Remove *E* from set *C* then randomly select an edge *R.* Add *R* to hypothesis *H*.

Quicksort

A comparison sorting algorithm is any that uses a comparison operation such as less than <, greater than >, equal to = or a combination of these to determine the order of two elements when sorting. Once sorted the list should possess these properties:

* Transitivity – If *a* *b* and *b c* then *a c*
* Totality – If *a* *b* or *b a* for all elements a and b

There are a large number of algorithms that fall under the category of a comparison sort and quicksort one of them. The quicksort algorithm was developed by Tony Hoare in 1959 and published in his paper “Algorithm 64 quicksort” (REF) in 1961. As the name would imply quicksort is one of the faster sorting algorithms with an average time complexity of *O* (*n* log *n*) when using the middle element as the pivot. However the algorithm’s worst case performance is *O* (*n*2) like many other sorting algorithms (REF), thankfully this is an unlikely case.

The quicksort algorithm consists of the following step:

1. Pivot *P* is an element selected from array *A*
2. Sort the array such that all elements smaller than the pivot are to the left of *P* and all elements larger than *P* are on the right. If an element is equal to *P* then it can be placed to the left or the right of *P* as long as this is kept consistent. Once complete *P* is in the correct location. This function is known as partitioning.
3. The array will now consists of 2 sub arrays, the first consisting of all elements smaller than the pivot and the second consisting of all elements larger than the pivot. Recursively run step 1 and 2 for the remaining sub-arrays.

Quicksort pseudo code:

**Algorithm 1**

1. **procedure** quicksort (Array *A*, int *lo*, int *hi*)
2. **if** *lo* < *hi* **then**
3. *P*  ← partition(*A, lo*, *hi*)
4. quicksort(*A, lo*, *P*)
5. quicksort(*A, P* + 1, *hi*)
6. **end if**
7. **end procedure**
8. **procedure** partition (Array *A*, int *lo*, int *hi*)
9. *pivot* ← (*lo* + (*hi* - *lo*) / 2)
10. *i* ← *lo* - 1
11. *j* ← *hi* + 1
12. **while *i* <= *j* do:**
13. **while** *A*[*i*] < *pivot* **do**:
14. *i* ← i + 1
15. **end while**
16. **while** *A*[*j*] > *pivot* **do**:
17. *j* ← j - 1
18. **end while**
19. **if** ***i* >= *j* do:**
20. **return** *j*
21. **end if**
22. **swap** *A*[*i*] with *A*[*j*]
23. **end while**
24. **end procedure**

Depth First Search

Depth First Search (DFS) is a traversal/searching algorithm that is commonly used for graphs and tree data structures. The concept for this algorithm was created in the 19th century by a mathematician by the name of Charles Pierre Trémaux as a method for solving mazes (REF). When used on a finite search space DFS is a complete algorithm as it performs an exhaustive search that will access every vertex (node) in a graph or all leaves in a tree. DFS is a greedy algorithm and will finish at the first goal located regardless of its optimality. The time complexity of DFS is dependent on the size of the graph and whether it is an implicit graph[[2]](#footnote-2) or explicit graph[[3]](#footnote-3). For an implicit graph traversal without repetition the time complexity is *O* (bd) where *b* is the branching factor[[4]](#footnote-4) searched and *d* is the depth searched. However the implementation of the graph will be and explicit version, as a result the time complexity becomes *O* (|*V*|+ |*E*|).

The DFS algorithm consists of the following steps:

1. Create a stack *S* and add (push) the starting vertex to *S.*
2. Run the step 3 and 4 until is *S* is empty.
3. Remove (pop) the last vertex *V* in *S*.
4. If *V* is not visited then label *V* as visited then push all adjacent vertices onto *S*.

DFS pseudo code:

**Algorithm 2**

1. **procedure** DFS (Stack *S*)
2. *V* ←**remove** last element from *S*
3. **add** *V* to visited set
4. **loop** through all edges connected to *V*:
5. *X* ← other vertex connected to *current* edge
6. **if** visited does not contain *X* **then**
7. *S* ← *S* **add** *X*
8. **end if**
9. e**nd loop**
10. DFS (*S*)

Kruskal’s algorithm

Kruskal’s algorithm is a greedy algorithm that aims to find the minimum spanning tree of a connected weighted graph. The algorithm was created by Joseph Kruskal in 1956 (REF) in the book ‘Proceedings of the American Mathematical Society’. It is considered to be one of the first efficient algorithmic solutions to the MST problem.

Kruskal’s algorithm consists of the following steps:

1. Graph *G* is the original graph given to the algorithm. Create a new edgeless graph *F* (all vertices are separate trees) from graph *G*.
2. Create a set *E* of all edges from *G* and order *E* in ascending order via weight.
3. Iterate through set *E* until end:

* Select the first edge *T* from *E* and add *T* to *F*.
* If *F* is a spanning tree return *F* and end algorithm.
* If *T* creates loop in *F* remove *T* from *F*.

Kruskal’s algorithm pseudo code:

**Algorithm 3**

1. **procedure K**ruskal (graph *G*)
2. *F* ← edgeless graph of *G*
3. *E* ← edge set from *G* sorted in ascending order
4. **loop** through edge set *E***:**
5. *T*← *current* edge from set *E*
6. *F* ← *F* add edge *T*
7. **if** *F* is a spanning tree **then**
8. *return**F*
9. **end if**
10. **if** *T* creates a loop in *F* **then**
11. *F* ← *F* remove *T*
12. **end if**
13. **end loop**
14. **end procedure**

Krushkal’s has an average time complexity of *O* (*E* log *E*) where *E* is the number of edges. The following is the analysis of the time complexity using a dis-jointed data structure:

* To create the edgeless graph takes constant time *O* (1).
* To creating a set of edges will take *O* (*V*) time.
* Sorting the edges using a comparison sort takes *O* (*E* log *E*) times as shown in quicksort.
* To iterate through each edges takes *O* (*E*) and to check for a union (loop) is *O* (*1*). As the check occurs inside the edge iteration the complexity becomes *O* (*E*).
* The final equation is *O* (1) + *O* (*V*) + *O* (*E* log *E*) + *O* (*E*)
* *E* can be represented as *V* \* (*V* -1) = *V*2 - *V* = *V*2 as this is the maximum number of edges possible. This means the largest term is *O* (*E* log *E*) thus becoming the time complexity

Stochastic Kruskal

For the generation of random spanning trees that will be used as the hypothesis of each agent a stochastic version of Krushkal’s algorithm has been created. This required a minor change to the selection process in order to add an element of randomness to this previously deterministic algorithm. The spanning trees generated by this graph are heavily dependent on the type of random distribution used for the selection/shuffling of edges.

Stochastic Kruskal’s consists of the following steps:

1. Graph *G* is the original graph given to the algorithm. Create a new edgeless graph *F* (all vertices are separate trees) from graph *G*.
2. Create a set *E* of all edges from the *G* and shuffle set *E*.
3. Iterate through set *E* until end:

* Select the first edge *T* from *E* and add *T* to *F*.
* If *F* is a spanning tree return *F* and end algorithm.
* If *T* creates a loop in *F* remove *T* from *F*.

Stochastic Kruskal pseudo code

**Algorithm 4**

1. **Procedure** StochasticKruskal (graph *G*)
2. *F* ← edgeless graph of *G*
3. *E* ← edge set from *G*
4. **loop** until *E* is empty**:**
5. *T*← **remove**randomedge from set *E*
6. *F* ← *F* **add** edge *T*
7. **if** *F* is a spanning tree **then**
8. **return** *F*
9. **end if**
10. **if** *T* creates a loop in *F* **then**
11. *F* ← *F* remove *T*
12. **end if**
13. **end loop**
14. **end procedure**

The time complexity of this algorithm is depends on a few different factors. One of these factors is whether the implementation is using a dis-jointed data set data structure which reduces the time complexity becomes *O* (*E*). This occurs because the time complexity for shuffling is *O* (*n*) where *n* is the number of elements. In the case of this algorithm *n* = *E* as *E* is the number of edges in the set. Using another data structure or method to detect a union (loop) would likely result in a larger time complexity.

# Project plan

What is TDD

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 class. It also helps break down the project into smaller more manageable sections 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 Github

# Testing and Specifications

To follow the development cycle the requirements and test have been created before the implementation of for each class. As the methods will be used in conjunction with each other, the tests have been designed to check the logic of each class and not the individual methods themselves. If the methods of a class were tested individually, the class could pass the unit tests but still cause errors in some of the other classes that use them.

The order of the following list corresponds to the order each class was created. This order is crucial as some classes are dependent on other classes to function. An example of this is the EWG class and its relationship with the Vertex and Edge class. The Edge and Vertex class can function completely fine without the EWG class in contrast to the EWG class which depends on both classes for the majority of its methods.

The following list is the requirements for each class:

Requirements for Edges:

* Must be connected on both sides as a vertex cannot be connected to nothing.
* The edge will not connect to a vertex if a self-loop will occur.
* The edge must be able to return both of the vertices connected for traversal.
* The edge orders the vertex classes before connecting to the edge. For example if the edge is being connected to vertex *B* and vertex *A*, vertex *A* will always be stored as the first variable.

Requirements for Vertex:

* The ability to add and remove edges to allow for the construction of the graph.
* 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 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.
* Edges cannot exist without connected vertices.
* Traversal between all connected vertices is possible.
* The graph can be deconstructed.

Requirements for Agents:

* An agent’s status is set to inactive by default.
* The agent is able to generate a random hypothesis.
* Hypotheses are spanning trees.
* The hypothesis must be accessible from external classes.

Requirements for DFS:

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

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.

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.

# Design and Implementation

Overall design

The java implementation has been constructed using elements of object oriented programming (OOP). The variables of the every class have been set to private then setter and getter methods have been created to incorporate encapsulation.

The first stage of the implementation was to create an edge weighted graph (EWG) as a framework for the hypothesis of the agents. An edge weighted graph consists of two main components which are represented in the implementation by an Edge class and a Vertex class. These classes create the foundation for the EWG (Edge Weighted Graph) class that can theoretically convey any undirected weighted graph. In order to traverse the graph both the Vertex class and the Edge class need to retain information which pertains to the other class. This resembles a “handshake” between classes which creates a strong chain of information thus making the program more robust and less prone to errors. To allow for the use of this implementation as a stepping stone for future development on the cost of the player’s network, an explicit graph has been implemented. This will allow the addition of a cost variable to the vertex class thus enabling an accurate recreation of the node map from the game.

After the framework had been created and tested the next stage of the implementation was to create the Agent class. The general skeleton for this class is very straight forward and simple, however the generation of random graphs for the hypothesis was far from simple. To generate the hypotheses a stochastic version of kruskal’s algorithm was developed, however this required the ability to detect loops within the graph. In order to achieve this, the DFS algorithm was implemented to traverse the EWG class. This algorithm was implemented as a separate class in order to store information about the EWG class which has been passed as an argument. This allows the information to be extracted by other classes and functions at later times in the program. Unfortunately using and implementation of the DFS algorithm will yield worst time complexities then a dis-jointed set data structure for both Kruskal’s and Stochastic Kruskal. However, this was unavoidable as the DFS class has much more utility than a dis-joined set would have due to the ability to record the edges of a cycle that occurs during the course of the algorithm.

Now the agent class has been completed and the framework is available the SDS algorithm can be implemented. The SDS class is the core of this project and took a considerable amount of time to implement all versions.

Vertex class

A vertex (node) is a point within the graph that represents an object; as such the vertex 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. It also doubles as a powerful tool for examining whether two Vertex classes are considered identical according to the graph. This is because when a graph is cloned, a new set of all Vertex and Edge classes is created. Now it is no longer possible to check if two Vertex classes are the same via the Object as the reference for these two classes will point to different locations in the memory. If these two classes store the same information they are considered the same vertex within the graph but not within the program.

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 this method will be used during the DFS traversal 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.

Edge class

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 associated with the edge
* String label – The name of the edge

An array has been used to store the two Vertex classes as the maximum number of vertex is predetermined to be two. A double is used for the weight variable to enable the used of fractions.

The class constructor has been overloaded to allow for 3 constructors. This allows for the representation of a weight-less graphs in addition to the EWG. 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. Within both of these constructors the init()[[5]](#footnote-5) method is called by passing both Vertex classes and a true boolean to the method to automatically connect the edge to both Vertex classes passed as arguments. The third constructor is used to bypass this automated connection for the creation of edges which may or may not be added to the graph.

The private method named init() has been created which takes two Vertex classes and a boolean as an argument. This method invokes the orderVertex() method to determine the order in which the Vertex classes will be stored via the label of the Vertex class. The Edge class is then labelled using a combination of the two labels from the Vertex classes. For example if the first class is labelled “1” and the second class is labelled “2” the Edge class’ label will be “12”. This is used to detect different instances of the same edge much like the Vertex class (IN REF). Next the init() method invokes the add() method of each Vertex class to add the newly created edge (this) to the edge sets of each Vertex class thus completing the “handshake” between these classes. This final step is determined by the Boolean argument passed to this class, if the Boolean is true then the final step is run. Otherwise the last step is not executed and the method is now complete. The access modifier of this method has been set to private to enforce encapsulation as this method is only required within the edge class itself.

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:

## (FIGURE)

EWG class

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 – Stores all Edge classes of the graph
* TreeMap<String, Vertex> vertices – Stores all Vertex classes of the graph

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 may be accessed to create the MST in the worst case scenario.

A TreeMap has been used to store all Vertex classes within 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 and the TreeMap 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 a EWG class as an argument and creates a clone of both the HashSet and TreeMap by iteration through the original EWG classes’ HashSet and creating new classes of all the Vertex classes using the addEdge() method. This in turn creates new classes of all the Edge classes, thus completing the cloning process. As the process creates completely new instances of all classes, the newly cloned EWG can be altered without affecting 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 which aid with the creation and deconstruction of graphs. In addition to this, methods for extracting information 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.

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, despite this all versions of this method feedback into the main addEdge() constructor. This allows any required logic to be enforced from one location and creates a synchronicity between all of the constructors.

The main addEdge(Edge e) constructor takes an Edge class as an argument then calls the chkEdge() method using the Edge class. The chkEdge() method uses the label of the argument edge to determine whether the edge already exists within the graph. This is done by iterating through the HashSet of edges and calling the String.equals() method to check if the label variables of each Edge class is the same. Next 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 twice 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. An Edge class is then created to connect the two Vertex classes using the double as the weight of the edge. The second variation of this method is used when creating weight-less 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 more 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 created both internally and externally from the graph and there is no strict order in which the vertices or edges must be added. The freedom which this design choice offers has 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.

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.

During the implementation of the Stochastic Krushkal algorithm it became apparent that the EWG would be required to have all Edges 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 to clear their HashSet<Edge> by using their removeAll() method. Once this has been completed the EWG HashSet<Edge> is cleared as well leading to a graph of only Vertex classes.

Agent class

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

* Boolean status – Represents if the agent is active or inactive
* EWG hypo – The hypothesis of the agent
* double fitness – The fitness associated with the agent

The agent class is much smaller and simpler than the other classes in this project. As a result the class has one main method called genHypo() which is called during within the constructor of the class. In addition to this the agent’s status is set to false as the test phase is required to activate agents. This can be seen as the default setting for the status Boolean. The fitness variable is left un-initialized as not all variations of SDS have a fitness for agents. This allow for the use of the agent class with outside of this project and allows the code to be more accessible for other developer who wish to implement SDS with this class.

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

The method called genHypo(EWG g) take an EWG class as an argument. This EWG is then used to generate a randomised spanning tree which is stored as the agent’s hypothesis. This method is an implementation of the stochastic Kruskal algorithm (IN REF).

It is important the note that the setter for the hypo variable creates a new EWG clone via the third constructor of the class.

DFS class

The depth first search algorithm has been implemented as a class in order to keep specific information that is used during Kruskal’s, genHypo() (stochastic Kruskal), and the update phase of SDS. As a result the following variables are used for DFS:

* HashSet<Edge> cycle – To store edges in cycle
* Boolean hasCycle – Records if graph has cycle
* Boolean isSpanTree – Records if graph is a spanning tree

The constructor for the DFS class has been overloads to allow for two ways to input the starting node for the DFS traversal. Both constructors take a EWG class as an argument. This argument is the graph that will be traversed. One constructor takes a String the other takes a Vertex class. This information is used at the starting point of the traversal. By overloading the constructor the class become able to use any information that can represent a Vertex class. This leads to a more flexible constructor overall. When a new DFS class is instantiated the haasCycle and isSpanTree Booleans are set to false and the search function is called.

The search function is broken into two parts using overloading. The first section is used to create variables and the second section runs a recursive DFS implementation. Part one takes a EWG and Vertex class as an argument to begin the setup. The following variables are created:

* HashSet<Vertex> visited – Set of all visited Vertex classes
* HashSet<Edge> used – Set of all traversed edges
* LinkedList<Vertex> frontier – List of all Vertex classes to be visited (functions like a stack)

SDS class

Kruskal class

Global class

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

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

# Research

# Results

# Conclusion

# Future Development

One of the major issues in this project is the time it takes to complete a full cycle of SDS on larger graph sizes. For example with the parameters of 200 agents and 500 iterations on a complete graph of 100 nodes the time taken to complete SDS became around 10 minutes. This became exponentially worse for a complete graph of 200 nodes as the time grew to over an hour.

This issue arose from the time complexity of Kruskal’s and Stochastic Kruskal’s using the DFS algorithm to detect loops. As the DFS algorithm takes place within an iteration of all edge the time complexity rises to *O* (*E*2). To fix this the dis-jointed data structure will need to be created to replace the DFS algorithm. This will drastically increase the performance as the methods in that will be used from this data structure operate in constant time *O* (1). In addition to this an implicit implementation for the graph will also help improve the speed of the program.

Another solution could be to implement and alternative hypothesis for each agent. An example of this would be to generate a number that is between 0 and the maximum number of spanning trees that are available in the graph. This way each number would represent a unique spanning tree for the graph. This method would require an implicit graph as the simplest way to implement this method would be to generate all the spanning tress of the graph and assigning each one a number before running SDS. If an explicit graph were to be used this could become a memory intensive approach which could cause errors with larger graphs.

# 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

1. 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). [↑](#footnote-ref-1)
2. An implicit graph uses representations of vertices and edges that are not stored as object within the computer’s memory [↑](#footnote-ref-2)
3. An explicit graph uses object to represent the vertices and edges of the graph. [↑](#footnote-ref-3)
4. The branching factor for the vertices in an undirected graph is *D* – 1 where *D* is the degree of the vertex. The vertex used as the starting point of the search is an exception to this rule with a branching factor of *D*. [↑](#footnote-ref-4)
5. The method name init is an abbreviation of initialise [↑](#footnote-ref-5)