SIMULATION OF OPTIMAL LOAD SHEDDING IN THE LIBYAN POWER GRID USING MULTI-OBJECTIVE GENETIC ALGORITHMS

Dissertation

NOVEMBER 23, 2015

MOHAMED SHARIF
H00158013

Abstract

Many developing countries suffer from power cuts that are caused by insufficient generated electrical energy to cope with the demand. This issue forces the energy companies to apply load shedding to keep the system in a stable state and avoid a total black out over the whole network. In this project, multi-objective genetic algorithms will be applied to tackle this problem and try to achieve a near optimal solution for load shedding in the Libyan power grid by trying to evenly distribute the load shedding over the full grid, prioritizing specific areas and avoiding duplicated load shedding as much as possible.

Declaration

I, Mohamed Sharif, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: …….

Date: ……/……/……...

Acknowledgements

I would like to thank my family for their support and help, all my friends who have supported me throughout all the struggles I had faced and who have helped me in overcoming them, Kevin Klein my Library friend who always motivated me and helped me push forward and finally to my supervisor who always gave me feedback and guided me through this project.

Table of Contents

Table of Figures

1 Introduction

Libya is a country located in North Africa and has one of the longest coasts shared with the Mediterranean Sea approximately 1800 km [1]. During the end of 2010, the Arab spring started as a chain reaction of revolutions from Tunisia to Egypt to some countries in the Arabian Peninsula and then finally arrived to Libya on 17th February 2011.

This caused most projects in the country to stop, which includes projects that were aimed at upgrading the power grid and installing new power stations all over the country to improve the electric generation to cope with the demand as the country developed. This was due to companies withdrawing their staff as a result of the unsafe situation in the country.

Now after 4 years the country has yet to achieve a stable state, has gone through many wars for power, and currently has two governments and two parliaments. One rules a small part of the eastern side of the country and has some small cities that support it in the western side of the country and the other rules the rest of the country. However, due to this unstable state none of the power grid related projects have resumed since then, this meant that the generation has not increased.

This caused the issue of not having enough generated electrical energy to cope with the demand, which caused GECOL (General Electric Company of Libya) a government owned company to use load shedding to keep the system from entering a complete black out over the whole country. At the current time, the load shedding is concentrated on the main major cities in Libya including but not limited to Tripoli and Misurata, and these can last to even 20 hours if not more a day with no power. This ad hoc method is unfair for those who live in these major cities.

This is considered a major problem that the people in Libya have to face in their daily lives, especially in this modern age, where almost all appliances rely on electricity as

their main power source. There is also the problem that even though the country is not in a stable state it is still developing and people are still building more houses which is adding more load onto the network that will cause the load shedding length to increase.

1.1 Aim

The aim of the project is to create a program that simulates the optimal load shedding over a high-level abstraction of the Libyan power grid while trying to distribute the load-shedding evenly across the grid as much as possible. This is because the methodology used now in the Libyan power grid distributes the load shedding in an unfair as it is concentrated on the major cities as well as it doesn't prioritise key areas such as locations where hospitals exist.

1.2 Objectives

The objectives of this project are:

1. To develop an accurate simulation of a high level abstraction of Libyan power grid.
2. To reduce the length that load shedding takes place in any area to a specific period no more than 2 hours were possible.
3. To develop a solution that carries out load shedding fairly between different geographical areas.
4. To ensure that high priority infrastructure (e.g. hospitals) remains supplied.
5. To investigate how Artificial Intelligent methods could be used to discover load shedding strategies that fulfil the above objectives.

2 Literature Review

The theoretical concepts of the techniques that are applied to achieve the project objectives are presented in this section. The literature review explains the major issue of this study, which is load shedding and then the artificial intelligence and its types would be outlined. Next Scheduling will also be discussed. Furthermore, previous works that were used to study load-shedding issue are discussed, in addition, reasons for choosing multi-objective GAs as the technique that will be used in this study are reviewed.

2.1 Load Shedding

Load shedding is the deliberate shutdown of part or parts of a power system to prevent the system from entering a complete failure state; this is resulted by the demand increasing beyond the capacity of the system [2].

2.2 Artificial Intelligence

Artificial intelligence (AI) can be defined as the “theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages” [3].

There are many techniques in AI, a brief overview would be provided about artificial neural networks, fuzzy systems and genetic algorithms as these techniques have been used previously in Load Shedding.

Artificial Neural Networks (ANN) is an AI model, which mimics the way the brain processes data. A model called a perceptron was created to model how a single neuron in the brain processes data. Multi-layer Perceptron comprise of three layer of neurons that can be categorised as input layer, hidden layer (not limited to one) and output layer. Every neuron is linked to all the neurons in the next layer and has a weight for the link, which is used to calculate the solution [4].

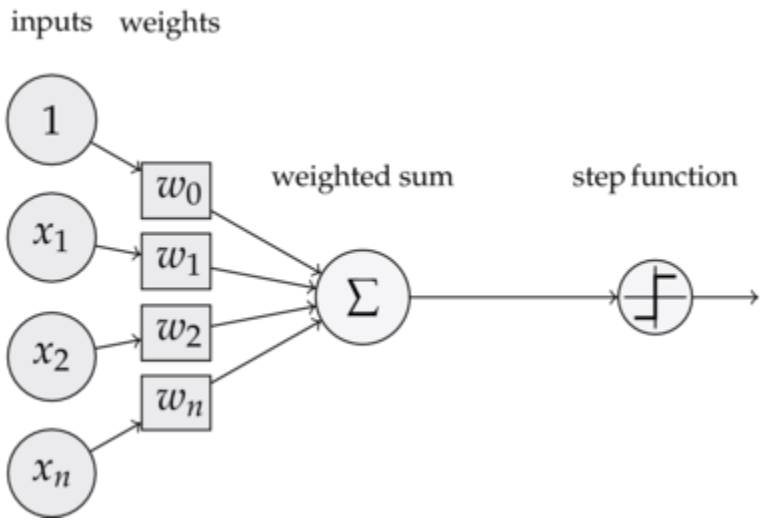

Figure 1 Perceptron [5]

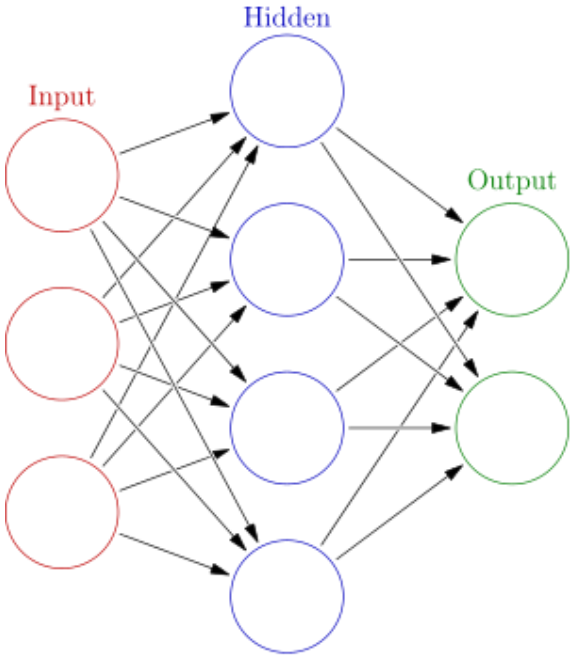

Figure 2 Example Artificial Neural Network [6]

Fuzzy Systems is an AI model, which can be used to solve problems where the answer is not expressible by “true” or “false” but rather “partially true” [4]. This is done using fuzzy sets theory in mathematics. A fuzzy set is a set were the members are assigned a value that refers to the degree of membership to that set as seen in the figure below.

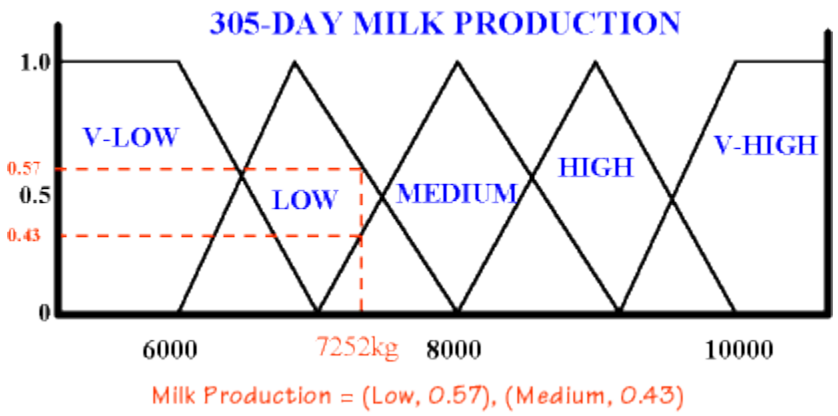

Figure 3 Example Fuzzy Set [7]

Genetic Algorithms is an AI model that mimics natural selection, it starts by creating a random population and then evolving that population by using genetic operators including crossover and mutation [4]. Finally, it assesses the fitness of each solution, chooses the top solutions and repeats the process until the solutions are relatively close and no better solutions are produced in a number of successive generations or it has found the maximum fitness or has passed the maximum amount of generations [4] (Figure 4).

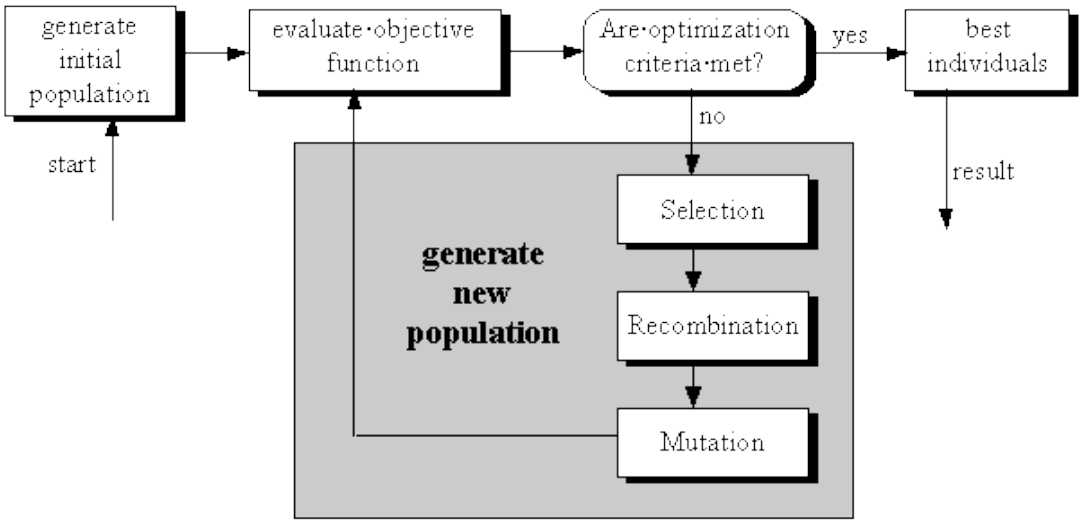

Figure 4 Genetic Algorithm Process [8]

2.3 Scheduling

In operating systems, scheduling is used to help share the systems resources as evenly as possible between running processes. This is done through multiple algorithms [9]; a modified version of these algorithms can be used in this project to help achieve the objective of fairer load shedding in the Libyan power grid. Round robin scheduling will be discussed because it is known for fair scheduling.

Round robin scheduling is a very simple widely used algorithm for scheduling [9]. It orders all the running processes into a list and processes the top of the list for a certain period called quantum, after the quantum runs out, the process is inserted at the end of the list and repeats the same procedure for every process [9]. This ensure fair access to the resource of the system.

2.4 Previous Work

In this section, the AI and scheduling techniques mentioned above and their application to solving load shedding and scheduling problems will be discussed.

2.4.1 Fuzzy Systems

Amit Jain proposed in 2010 a fuzzy system that could predict a day ahead electric demand, this was achieved using Mahalanobis distance [10]. The method used, relied on searching previously gathered data of the system that a day ahead prediction is required for, this is done by comparing a 24 hour temperature, humidity and also a day type variable to evaluate the similarity with previous data. This is then combined with a correction factor to give an accurate result. This was tested in MATLAB using 7-months' worth of data [10].

This can be used to help deal with the increased load before it occurs using methods such as reducing the voltage [10]. The method discussed above will not be incorporated into this project, as it is out of the scope of this project but can be used to extend the project in the future.

2.4.2 Artificial Neural Networks

In 2005 Cheng-Ting Hsu proposed an artificial neural network, which solves the load-shedding problem, however this solves it from a different aspect, the purpose of his solution is to find the actual proper minimum load that requires shedding. This is accomplished by combining the frequency and other selected features from data supplied by Taipower a Taiwanese power company. Using the data with the artificial neural network that uses Levenberg–Marquardt back-propagation algorithm, the algorithm was able to find the optimal load to be shed [11]. The results showed that the optimal load to be shed was 2521 MW and 1950 MW in the northern-central subsystem during peak and off-peak periods respectively compared to the previously found result of 5094 MW and 3182 MW respectively, which is considered at least about 30% less [11].

In this project, only the load would be considered and we will not go into the details regarding the frequency and other data, which has effect over the amount of load to be shed as it is out of the scope of this project but can be used to extend this project in the future.

2.4.3 Genetic Algorithms

Rao, K.U. (2013) introduced a solution to the load shedding problem using a genetic algorithm. In this algorithm a method was introduced to prioritise specific loads, these loads were sorted into categories [12]:

- Domestic Loads
- Commercial Loads
- Industrial Loads
- Public utilities
- Critical Loads

Using these categories, a high priority is assigned for each load category at a specific time of day, where each load of this category has a higher priority compared to other loads to try to force the algorithm away from choosing that load as the solution [12]. The representation of the chromosome that was used was a binary {0, 1} to represent if the load is shed or not.

In this project, a slightly modified version of the time-based priority will be incorporated. Furthermore, the chromosome representation will also be used in this project.

2.4.4 Weighted Round Robin Scheduling

Weighted round robin scheduling algorithm is a further development of the round robin algorithm. Weighted round robin scheduling algorithm tackled one of the problems that the round robin algorithm suffered from, which does not differentiate between task priorities [13]. In any operating system, there are tasks that have a higher priority compared to others for the system and the user.

Weighted round robin scheduling algorithm tackled this issue by assigning weights to each task in the list of processes, such that when the list is ordered higher priority tasks are put at the front of the list [13]. Then the process continues similar to the regular round robin algorithm, by adding the task at the end of the list after it time on the CPU runs out.

Weighted round robin scheduling algorithm also added a small modification to the time-share allowed on the CPU such that higher priority tasks have a higher time-share while also not surpassing a certain threshold and for the lower priority tasks less time while not going lower than a certain threshold [13]. This is to help prioritize the tasks with higher priority while also avoiding starvation in the system. Starvation is the act of tasks not being allocated a time-share on the CPU because other task is scheduled due to having a higher priority. Weighted round robin scheduling algorithm sacrifices efficiency to achieve fairness and system responsiveness [13]. After running a

simulation in C# that compares between weighted round robin and round robin scheduling algorithms, it was found that weighted round robin performed more efficient in many cases [13].

The concept of using a list in the weighted round robin scheduling algorithm can be used in this project such that the list is initially empty and for every time load shedding occurs the area that has been shed is added to the list. Later the algorithm used in this project will be issued a penalty based on the position of the area chosen in the list (similar to how weighed round robin scheduling assigns weights) to try to force it away from choosing that area as a solution. Using this method will help in achieving a fairer solution to load shedding in the Libyan power grid.

The concept can also be used to sort out the prioritized areas such as areas where hospitals exists, which load shedding must not be applied to if alternatives exist and areas where important meeting occurs, which will be only assigned a priority for a period of time and then removed from the list.

Multiple list can be used to divide the priorities into different categories, based on which list an area is located in a penalty will be applied to the algorithm, these are possible categories for the lists:

- High Priority (areas where hospitals exist).
- Medium Priority (areas where government organization exist).
- Low Priority (areas that have an economic importance).
- Non-Prioritized (residential areas).

The reasons behind this prioritization are the following

- Medical: as these areas are critical areas which if they are treated normally may cause lose in life.

- Government Organization: as these areas provide necessary services that are required by the people and also important meeting occur in these areas, since GECOL is a government based company these areas need to be prioritized.
- Economical: as these areas help grow the economy of the country, unlike some countries GECOL is the only provider so the reason for prioritizing these areas is not to compete in the market.
- Residential: as these are the remaining category and has a lower priority, they are considered not to have any priority as they do not include any life threating or economic importance.

2.4.5 Multi Objective Genetic Algorithms

Multi-objective genetic algorithms were used to solve scheduling problems in computer grids where scheduling is very important to achieve lower running costs and faster execution time.

Ritu Garg proposed the usage of multi-objective genetic algorithms in 2011, his study proposed the modification of the NSGA-II algorithm into the R-NSGA-II which stands for reference point non dominant sorting genetic algorithm [19]. The usage of the reference point operator replaced the crowding operator to achieve solutions in areas of interest in the Pareto optimal front. The pseudo code of how the reference point operator is calculated can be seen in the figure below.

1. Assign rank to each solution with respect to each reference point according to calculated Euclidean distance (Equation 4).
2. Determine preference distance of each solution by selecting minimum rank with respect to all reference points.
3. Make groups of solutions by applying ε-clearing idea and retain one random solution from each group.

Figure 5 Reference Point Operator [19]

It was proposed that the reference point is placed by the user to help encourage the algorithm to search the problem space in areas of interest of the user [19]. This showed efficient results which an example can be seen below where the algorithm was compared to the original NSGA-II to solve the same problem.

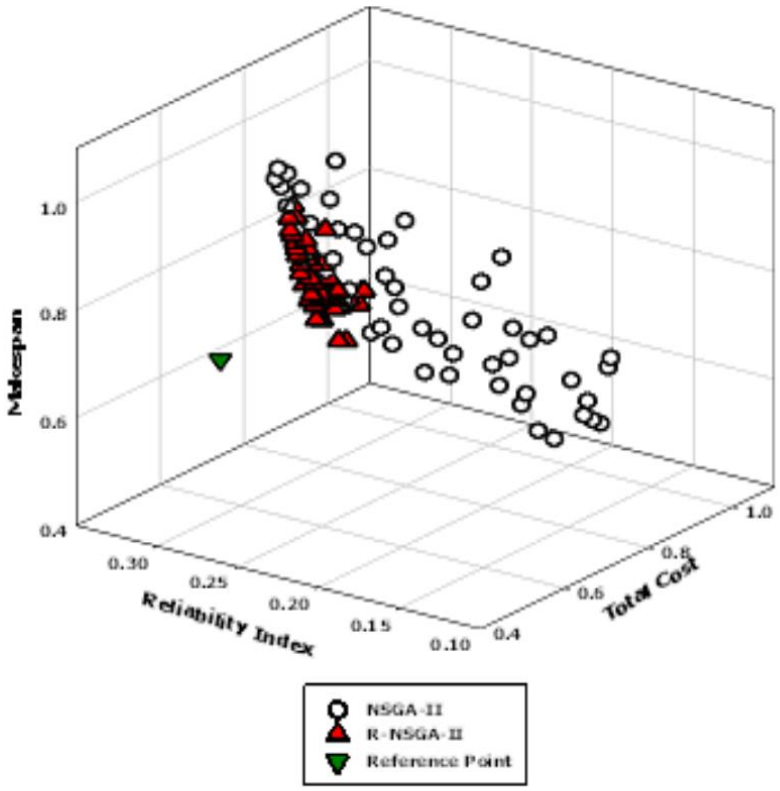

Fig. 4: Obtained Pareto Optimal front on intermediate constraint

Figure 6 Example R-NSGA-II vs NSGA-II Solving Scheduling Problem [19]

Another example of the usage of multi-objective genetic algorithms in scheduling problems was the usage of a hybrid species based multi-objective evolutionary algorithm (SMOEA) introduced by Hongfeng Wang [20].

In this algorithm the species modal was used, where sub populations are used to produce solutions for sub problems. Then when offspring are created, individuals from different sub populations are used.

Using this algorithm results showed that it surpassed both the NSGA-II and the MOEA/D algorithms in solving the flow shop scheduling problem which is a known multi-objective scheduling problem (MOSP) [20].

This algorithm will not be used in this project due to time constraints but can be investigated in the future to help optimise the current implementation.

2.5 Genetic Algorithms

Genetic Algorithms will be used in the development of the application in this project for the following reasons:

- Having a population, which allows the problem space to be searched diversely.
- Multi objective support, which will help achieve the objectives of this project that in some cases contradict each other.
- Control of how long it runs for by limiting the number of generations.

2.5.1 Multi Objective

Multi objective genetic algorithms are genetic algorithms where the problem that requires solving has more than one objective to accomplish; often these objectives may contradict each other and in this case, the optimal solution would be a solution that tries to satisfy all of the objectives, this is called the Pareto-optimal solution [14]. PESA-II and NSGA-II will be discussed below as one or both of them will be used to complete this project. PESA-II was chosen due to the recommendation of the supervisor and NSGA-II was chosen because it has been shown to solve many problems better than Multi objective genetic algorithms [14].

2.5.1.1 NSGA-II

NSGA-II algorithm was developed by Kalyanmoy Deb in 2002, which stands for non-dominated sorting algorithm [14]. NSGA-II was an improvement of NSGA due to the following problems it had tackled [14]:

- The high complexity of NSGA which was $O(MN^3)$
- Lack of elitism.
- Need for specifying the sharing parameter.

Below are the steps this algorithm follows for each generation [14] (Figure 5):

1. Create offspring population (Qt) using selection, crossover and mutation and combine with parent population (Pt).
2. Sort the population (Rt) into non-dominated fronts (F1, F2, F3).
3. Calculate crowding distance.
4. Sort solutions in descending order (Pt + 1).

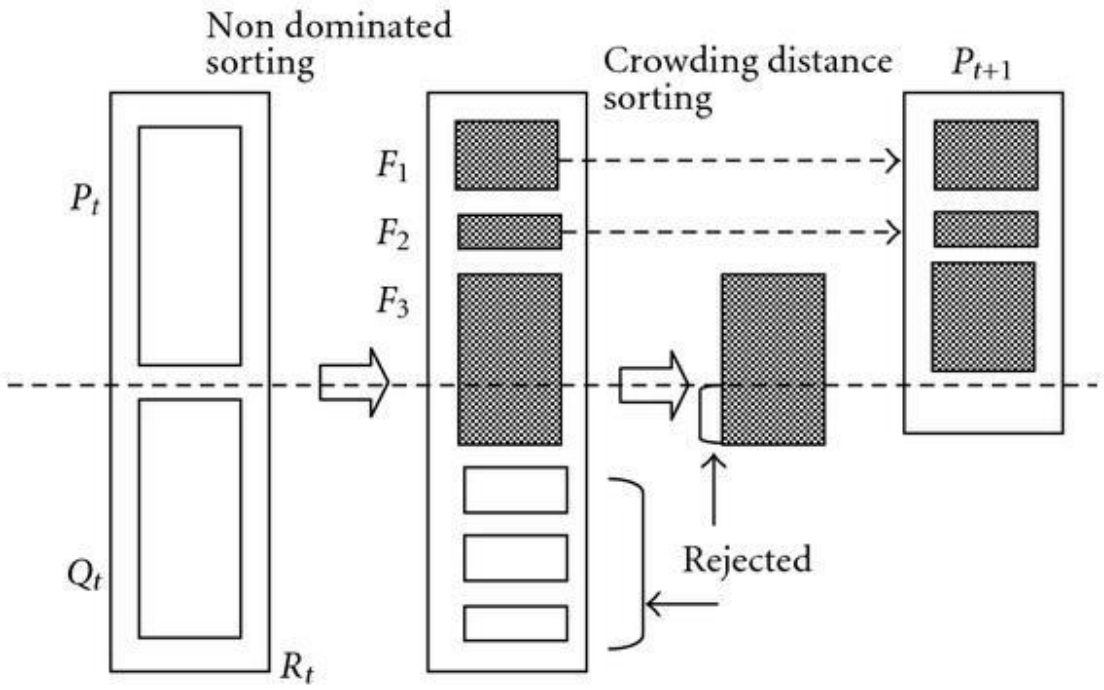

Figure 7 NSGA-II Algorithm Process [15]

2.5.1.2 PESA-II

In 2001, David Corne developed PESA-II algorithm, which stands for Pareto Envelope base Selection Algorithm [16]. This was an improvement on PESA by reducing the complexity, which was achieved by utilizing region-based selection compared to individual based selection [16]. In region-based selection the solution space is divided into hyper-boxes were the hyper-box represents a region and the region is used in helping to find diverse solutions as shown in the figure below where hyper-box C would be a better solution than hyper-box B to achieve a varying set of solutions [16]. This will be implemented in the case difficulties are encountered when trying to use a

framework which supports NSGA-II and not succeeding in implementing the NSGA-II algorithm.

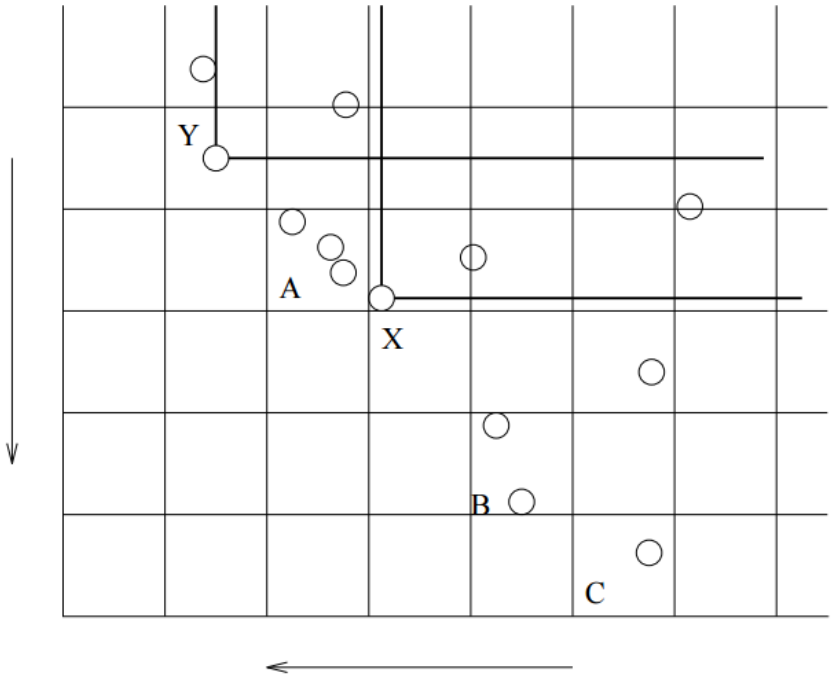

Figure 8 Hyper-Box Example [16]

Below are the steps this algorithm follows each generation [16]:

1. Add non-dominant solutions for the internal population to the external population.
2. If a termination condition has been satisfied then stop and return the external population, otherwise delete contents of the internal population.

2.6 C++

C++ is a programming language that was originally released in 1985 by Bjarne Stroustrup. It was a further development of the C language with the added functionalities of object oriented design [17]. C++ is currently ranked third according to the IEEE spectrum ranking and is currently used in many real-time and embedded systems.

This project will be developed in C++ for the following reasons:

- C++ is fast this means it can calculate the solutions in a relatively fast time in this time critical problem, were time is limited to make a decision before the system fails.
- C++ has support for object-oriented designs this will help simplify designing the power grid model.
- C++ support generic programming this will help in creating a generic data structure for the power grid model.

3 Methodology

This project can be divided into three main processes. First converting the data provided by GECOL into machine readable format that the software can read and process. Then creating a model of the Libyan power grid and finally creating a bio-inspired controller that will evaluate the problem and return an evolved solution.

3.1 Converting The Data

The data was first analysed to decide what is the best option to convert the data, which was then converted.

3.1.1 Data Analysis

In this section the data supplied by GECOL, which will be used to help accomplish the aims and objectives of this project will be discussed. However, this data will not be available because it is confidential data and was only provided by GECOL to help in accomplishing the objectives of this project. Some of this data is also considered a security risk as it shows the design of the grid, which could be used by terrorists/vandals to plan blackouts and cause disturbance in the country especially in its currently unstable state.

3.1.1.1 Power Grid Designs

The power grid designs supplied by GECOL, which is in pdf format can be divided into two sections:

- High-level power grid design.
- Low-level city based power grid design.

3.1.1.1.1 High Level Power Grid Design

This design shows a high-level abstraction of the Libyan power grid, this includes positions of power plants, High-voltage powerline cable paths and powerline voltage values.

3.1.1.1.2 Low Level City Based Power Grid Design

These designs display low-level designs of how areas in Tripoli and Misurata are connected and which area is connected to which substation.

3.1.1.2 Historical Data

This data shows what areas in the Libyan power grid were on at a specific time combined with their voltage and load and the areas that load shedding was applied to at that time as recorded by GECOL. Currently one hour of data has been received for two of the major cities in Libya, Misurata and Tripoli, which were chosen as they suffer from load shedding the most. This data is in excel format.

3.1.2 Conversion Process

3.1.2.1 Power Grid Designs

The data provided was in pdf format this posed a difficulty in converting the data to machine readable format using software. This was the reason for deciding to do the conversion process manually.

The data was converted into xml format and was stored in multiple files, “areas.xml”, “powerstations.xml”, “substations.xml” and “regions.xml”. This would help in scaling up the project to include all substations and areas in the Libyan power grid.

The regions of the grid were taken from a document released by GECOL and were then stored in xml format where the name and an ID was assigned to each region as seen in the example below of the Tripoli region.

```
<region>
	<id>1</id>
	<name>Tripoli</name>
</region>
```

The power stations, their capacity and the regions which they are in were provided during a telephone discussion. An example of how a power station is stored can be seen below.

```
<powerstation>
	<id>1</id>
	<name>Alkhumus Steam 1</name>
	<capacity>120</capacity>
	<region>1</region>
</powerstation>
```

The substations were provided in the low level design documents as they are separated by substations. An example of how a substation is stored can be seen below.

```
<substation>
    <id>1</id>
    <name>West Tripoli</name>
    <region>1</region>
</substation>
```

The areas were provided in the low level designs were each substation has areas connected to it. The ids for the areas were assigned according to their order in historical data provided to speed the conversion process for the historical data. An example of an area can be seen in the figure below where the area which its name is inside the yellow box will be found in the historical data, assigned the id and the substation id is assigned according the id of the corresponding substation in the "substations.xml" file.

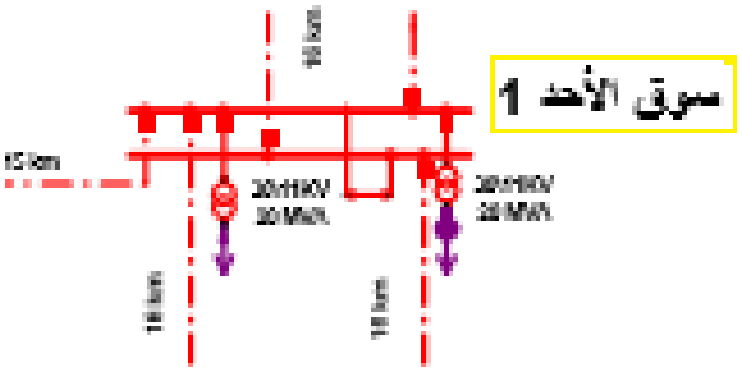

Figure 9 Example of Designs

Every area is assigned a type based on category it belongs to Residential, Economical, Governmental and Medical. An example of how an area was stored can be seen below.

```
<area>
    <id>1</id>
    <name>Aliskan-1</name>
    <substation>3</substation>
    <type>Residential</type>
</area>
```

3.1.2.2 Historical Data

The historical data was provided in excel format with a lot of unnecessary data for the implementation. An example of the data can be seen in the figure below where the yellow highlighted field is the only data kept since all the areas are sorted according to the order in which they are ordered in the excel document. This was then stored into a txt file.

19.5	10.5	20	(1) رقم 11/30	الاسكان	1
	9	15	(2) رقم 11/30		

Figure 10 Example of Historical Data

3.2 Grid Model

3.2.1 Data Structure

A tree data structure was used to develop the grid model; this was found to be an appropriate data structure due to the following reasons:

- Maintaining the overall structure of the power grid.
- Reducing the complexity of the power grid.
- Simplifying the process of updating the grid in the simulation.

The first design of the power grid model can be seen in the figure

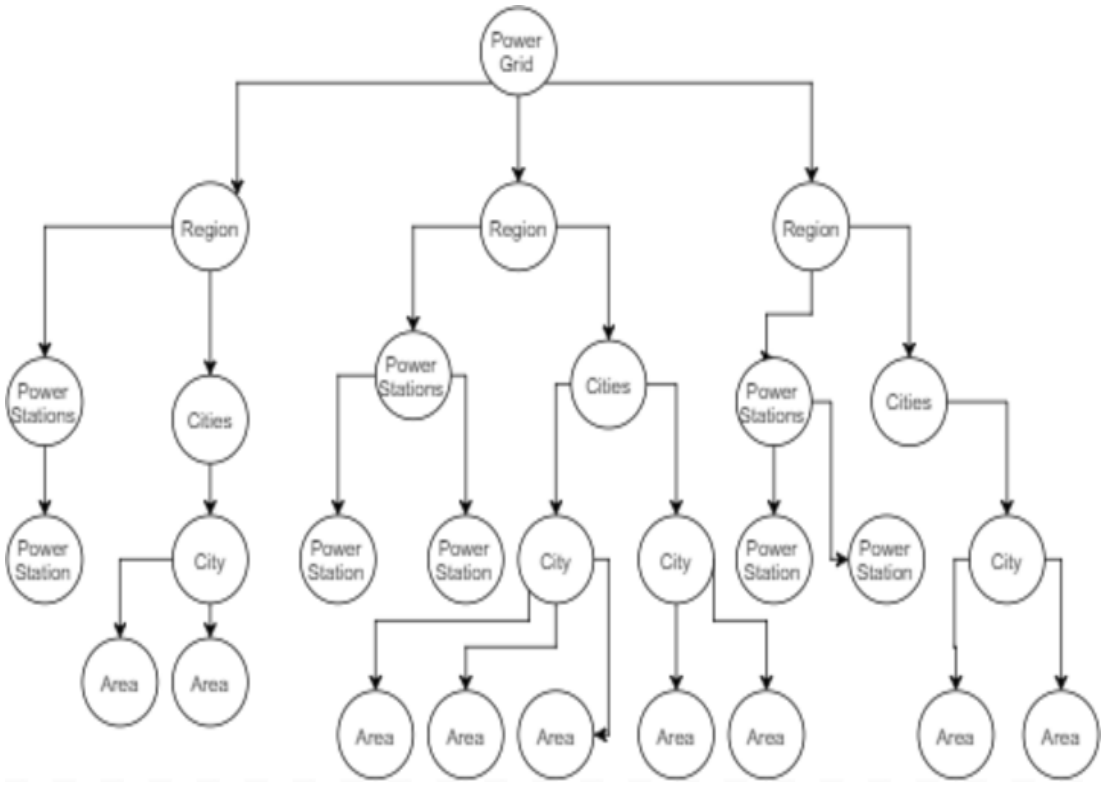

Figure 11 Power Grid Tree Model First Design

After careful consideration it was decided to change the design for the following reasons:

- Removing the nodes such as “Power Stations” and “Cities” reduces the number of operations required when updating the values of the power grid to speed up the process.
- Dividing into substations rather than cities is a better decision as it can also help the operators in tracking any issues while monitoring the power grid for maintenance purposes.

The new design can be seen in the figure below.

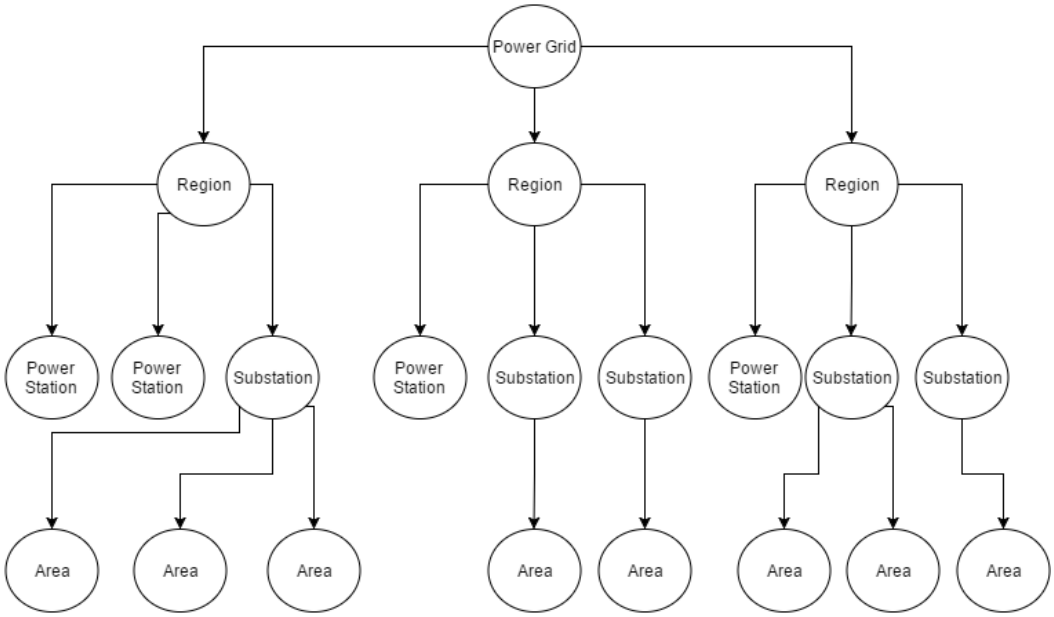

Figure 12 Power Grid Tree Model Final Design

3.2.2 Design Pattern

The Publisher/Subscriber pattern was used when building the grid model, where a group of object are considered as publishers that publish information to the other group of objects which are subscribers if they are subscribed to the publisher object.

In the grid model each parent node is subscribed to all its child nodes to receive updates as the simulation is running.

3.3 Bio-Inspired Controller

It was decided that a bio-inspired controller will be implemented to search the problem space for a near optimal solution. NSGA-II is the specific algorithm implemented in this project which is a multi-objective genetic algorithm (refer to section 2.5).

A framework was not used as planned due to difficulties encountered when using them, this was mainly because the documentation provided with them was not sufficient. NSGA-II was implemented from scratch using C++ specifically for this project.

Below is a list of fitness functions that were used to achieve the requirements of the controller.

1. F (x) = (area load – x), where “x” is the load to be shed.
2. F (x, previous) = IF x not in previous return -1;

   ELSE return 10 * position in previous;

   Where “previous” is a list of all loads that have already been shed.
3. F (x) = IF x has priority return penalty;

   ELSE return 0;

Overall getting the smallest fitness is the optimal result. These fitness functions can still be changed later to achieve better results if required.

4 Implementation

In this chapter the requirement for the implementation will be discussed as well as the process of how the grid and bio-inspired controller were implemented.

4.1 Requirements Analysis

This section will discuss the functional and non-functional requirement of the system and assign priorities to each one.

4.1.1 Functional Requirements

Non-Duplicate Shedding **Priority: High**

To not apply load shedding to the same area twice as much as possible in a chosen period this is to achieve a fairer distribution of load shedding in the power grid.

Prioritize Areas **Priority: Medium**

To have priorities for important areas such as building where important meetings occur in hospitals, which should be avoided as much as possible as they are critical areas.

Best Fit Shedding **Priority: Medium**

To choose the area that best fits the load that is required to be shed when possible, this is to help minimise the loss of generated energy.

Visual Display of Simulation **Priority: Optional**

To provide a visual display of the load shedding simulation, this is to help understand how the simulation is working in a visual form.

Day Ahead Prediction **Priority: Optional**

To use the suggested fuzzy system in the previous work section with the Libyan power grid, this would help in planning for actions such as reducing voltage to keep the system in a stable state.

4.1.2 Non-Functional Requirements

Cross Platform **Priority: Medium**

The application should have the ability to run on Windows and Linux operating systems, these two operating systems were chosen because they are most likely to be used by GECOL, which would help in providing a system that can be deployed by GECOL.

Framework Usage **Priority: Medium**

To use open beagle as a multi-objective genetic algorithm framework to implement the bio-inspired controller, this is to simplify the process of using AI techniques in this project.

4.2 Grid Model

The grid model was implemented in three phases discussed in detail below.

4.2.1 Tree Structure

The tree structure was implemented in five classes representing the nodes in the design in section (3.2.1).

Grid class: this stores the overall load and generation of the power grid which can be used as the entry point of the bio inspired controller.

Region class: this stores the details of a power station, e.g. id, name, overall generation and overall load of the region.

Power station class: this stores the details of a power station, e.g. id, name, generation and capacity.

Substation class: this stores the details of a substation, e.g. id, name and overall load.

Area class: this stores the details of an area, e.g. id, name and load.

The structure is built by first creating a grid object then adding in the regions as children of that object. Then power stations and substations are added as children to the regions based on their id. Finally, areas are added as children to the substations based on their id.

4.2.2 Pattern

The Publisher/Subscriber pattern was modified in this project as a publisher only publishes data to one subscriber, for this reason there was no need to store a vector of subscribers but rather just one pointer to the subscriber.

All the middle nodes in the tree structure (regions and substations) are both Publishers and Subscribers at the same time, where they receive data from their children and publish to their parents.

Since in our case data was published from the leaf nodes up, every node stored a pointer for its parent which is used to call the update function with the required data when new data is published.

4.2.3 Simulation

The xml data was loaded into the program using a static library called pugixml. Using this data objects were created from the classes representing the data and where stored into vectors as seen in the example below.

```
std::string source = "regions.xml";
// create an xml document object then load file
pugi::xml_document doc;
if (!doc.load_file(source.c_str())) return -1;

std::cout << "parsing regions" << std::endl;
pugi::xml_node regions = doc.child("regions");
// loop through the children of the node and de serialize data into object
for (pugi::xml_node reg = regions.first_child(); reg; reg = reg.next_sibling())
{
	region r;
	r.id = atoi(reg.child("id").child_value());
	r.name = reg.child("name").child_value();
	r.gridPointer = &::g;
	::regions.push_back(r);
}
```

The tree structure was then created by pointing to the objects stored in the vectors. The main reason for this was to speed up the operations that are applied to the same type of objects by applying them directly through the vectors without having to search down the tree and then applying them as seen in the example below.

```
// insert substation as children of the region they belong to by storing a
pointer to them in the substations vector
for (int f = 0; f < ::subStations.size(); f++)
{
	subStation s = ::subStations.at(f);
	g.regions.at(s.regionPointer->id - 1)->
subStations.push_back(&::subStations.at(f));
}
```

The grid model was then initialized with the load data which was previously converted into the loads.txt file as a starting point for the simulation as seen below.

```
// initialize the loads
std::ifstream stream("loads.txt");
std::string line = "";
for (int l = 0; l < ::areas.size(); l++)
{
	std::getline(stream, line);
	::areas.at(l).init(atof(line.c_str()));
}
stream.close();
```

After that the amount of energy generated by each power station was initialized with 80% of the power station's capacity, this was decided after discussing with people from GECOL and learning that on average the power stations run at that percentage.

In both cases described above a chain reaction was invoked to update the values stored for both the generation and load in the grid model nodes due to the Publisher/Subscriber pattern where each node will call its parent update method to update the values as seen in the example below.

```
void region::update(double l, double g)
{
	load += l;
	generation += g;
	// call the update function of parent with the published data
	gridPointer->update(l, g);
}
```

Finally, the simulation enters an infinite loop which has a timer to delay the reiterating of the loop. In this loop the grid is updated with changes in values in the generation and load with a random value between a 10% decrease and 10% increase as seen in the example below.

```
int l = 0;
if (load != 0)
{
	srand(time(NULL) * std::rand() * 5);
	l = -(load * .1) + fmod(std::rand(), (load * .1) * 2);
	previousLoad = load;
	load += l;
	// to keep the simulation running with a higher chance of shedding occurring the load has a higher and lower bound.
	if(load < baseLoad * .8)
	{
		load = baseLoad * .8;
		l = load - previousLoad;
	}else if(load > baseLoad * 1.4)
	{
		load = baseLoad * 1.4;
		l = load - baseLoad;
	}
	subStationPointer->update(l);
}
```

4.3 NSGA-II

4.3.1 Implementation

The NSGA-II algorithm was implemented in three different classes individual, population and nsga2.

Individual class: this represents one individual possible solution.

Population class: this holds a populations of individuals and also stores the non-dominant fronts.

NSGA2 class: this holds all the logic behind the NSGA-II algorithm and the evolution process that returns the solution.

The pseudo code which gives the overall structure of how the algorithm is implemented can be seen in the figure below.

$R_t = P_t \cup Q_t$ — combine parent and offspring population
$\mathcal{F} = \text{fast-non-dominated-sort}(R_t)$ — $\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \ldots)$, all nondominated fronts of R_t
$P_{t+1} = \emptyset$ and $i = 1$
until $|P_{t+1}| + |\mathcal{F}_i| \leq N$ — until the parent population is filled
  crowding-distance-assignment$(\mathcal{F}_i)$ — calculate crowding-distance in $\mathcal{F}_i$
  $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$ — include ith nondominated front in the parent pop
  $i = i + 1$ — check the next front for inclusion
Sort$(\mathcal{F}_i, \prec_n)$ — sort in descending order using $\prec_n$
$P_{t+1} = P_{t+1} \cup \mathcal{F}_i[1 : (N - |P_{t+1}|)]$ — choose the first $(N - |P_{t+1}|)$ elements of $\mathcal{F}_i$
$Q_{t+1} =$ make-new-pop(P_{t+1}) — use selection, crossover and mutation to create a new population Q_{t+1}
$t = t + 1$ — increment the generation counter

Figure 13 NSGA-II Algorithm [15]

The pseudo code that explains how the sorting part of the algorithm works can be seen in the figure below.

fast-non-dominated-sort(P)
for each $p \in P$
  $S_p = \emptyset$
  $n_p = 0$
  for each $q \in P$
    if $(p \prec q)$ then — If p dominates q
      $S_p = S_p \cup \{q\}$ — Add q to the set of solutions dominated by p
    else if $(q \prec p)$ then
      $n_p = n_p + 1$ — Increment the domination counter of p
  if $n_p = 0$ then — p belongs to the first front
    $p_{\text{rank}} = 1$
    $\mathcal{F}_1 = \mathcal{F}_1 \cup \{p\}$
$i = 1$ — Initialize the front counter
while $\mathcal{F}_i \neq \emptyset$
  $Q = \emptyset$ — Used to store the members of the next front
  for each $p \in \mathcal{F}_i$
    for each $q \in S_p$
      $n_q = n_q - 1$
      if $n_q = 0$ then — q belongs to the next front
        $q_{\text{rank}} = i + 1$
        $Q = Q \cup \{q\}$
  $i = i + 1$
  $\mathcal{F}_i = Q$

Figure 14 Fast Non-Dominated Sorting [15]

In this project the implementation of how an individual dominates another individual can be seen below where the individual with the smaller accumulated result is the dominant one.

```
bool Individual::dominates(Individual * o)
{
	int result1 = 0;
```

```
        int result2 = 0;

        for (int i = 0; i < 3; i++)
        {
                result1 += objectives.at(i);
                result2 += o->objectives.at(i);
        }
        if (result1 <= result2)
                return true;
        else
                return false;

}
```

After the individuals are sorted into non dominant fronts, all the individuals in each front is then sorted according to their crowding distance. The pseudo code for how the crowding distance is calculated can be seen in the figure below.

crowding-distance-assignment($\mathcal{I}$)
$l = |\mathcal{I}|$ — number of solutions in $\mathcal{I}$
for each i, set $\mathcal{I}[i]_{\text{distance}} = 0$ — initialize distance
for each objective m
  $\mathcal{I} = \text{sort}(\mathcal{I}, m)$ — sort using each objective value
  $\mathcal{I}[1]_{\text{distance}} = \mathcal{I}[l]_{\text{distance}} = \infty$ — so that boundary points are always selected
  for $i = 2$ to $(l - 1)$ — for all other points
    $\mathcal{I}[i]_{\text{distance}} = \mathcal{I}[i]_{\text{distance}} + (\mathcal{I}[i+1].m - \mathcal{I}[i-1].m)/(f_m^{\max} - f_m^{\min})$

Figure 15 Crowding Distance [15]

The individuals are assigned a value for each of the three objectives, the objectives are best fit, previously shed and priorities respectively (refer to section 3.3). The implementation of how they are calculated can be seen below.

```
void NSGA2::calculate_objectives(Individual & individual)
{
        individual.objectives.clear();
        individual.objectives.push_back((load                                   -
std::accumulate(individual.features.begin(), individual.features.end(), 0)) -
load_to_be_shed);
        std::vector<int> sheded;
        for (int i = 0; i < individual.features.size(); i++)
        {
                if (individual.features.at(i) == 0)
                        sheded.push_back(i);
        }
        double pen1 = 0;
        for (int i = 0; i < sheded.size(); i++)
        {
                for (int j = 0; j < previously_sheded.size(); j++)
                {
                        if (sheded.at(i) == previously_sheded.at(j).id - 1)
```

```
                        pen1 += 50 * (j + 1);
                    else
                        pen1++;
            }
        }
        individual.objectives.push_back(pen1);
        double pen2 = 0;
        for (int i = 0; i < sheded.size(); i++)
        {
            for (int j = 0; j < areas.size(); j++)
            {
                if (sheded.at(i) == areas.at(j).id - 1)
                {
                    if (areas.at(j).area_type == "Economical" && times
> 8 && times < 18)
                        pen2 += 20;
                    if  (areas.at(j).area_type  ==  "Governmental"  &&
times > 8 && times < 15)
                        pen2 += 40;
                    if (areas.at(j).area_type == "Medical")
                        pen2 += 1000000;
                }
            }
        }
        individual.objectives.push_back(pen2);
}
```

The explanation of the reasons the fitness functions described above use these values are:

- Best fit, it is assigned the amount that will be shed minus the amount that is required to be shed. This is to ensure that the lower the value is the better.
- Previously shed, it is assigned a penalty of 10 times the position in the vector for every area that has been previously shed otherwise the value would be decreased by a value of 10. This is to promote the solutions which have less areas shed at one time and in the case an area has been shed before it will promote the area that have been shed at an older time.
- Priorities, it is assigned a penalty based on the type of area:
    - Economical are assigned a penalty of 20 if the time is between 8am to 6pm, this is because that is working times for businesses in Libya.
    - Governmental are assigned a penalty of 40 if the time is between 8am to 3pm, this is because that is working times for governmental locations in

Libya. It has a higher value than Economical areas to elevate the solutions which have economical areas rather than governmental.

- Medical are assigned a penalty of 1000000 at any time to try and force the algorithm to favour solutions which do not have medical areas scheduled to be shed.

The solution chosen from the Pareto-optimal front is selected by the usage of two objectives out of the three, priorities and previously shed:

- The solutions in the Pareto-optimal front are sorted based on the better previously shed objective.
- The top 30% of the sorted solutions are then selected.
- The top solutions based on the priorities objective is chosen from the selected solutions.

An example can be seen in the figure below where the orange area refers to the selected 30% and the green area refers to the solution selected based on the priorities objective.

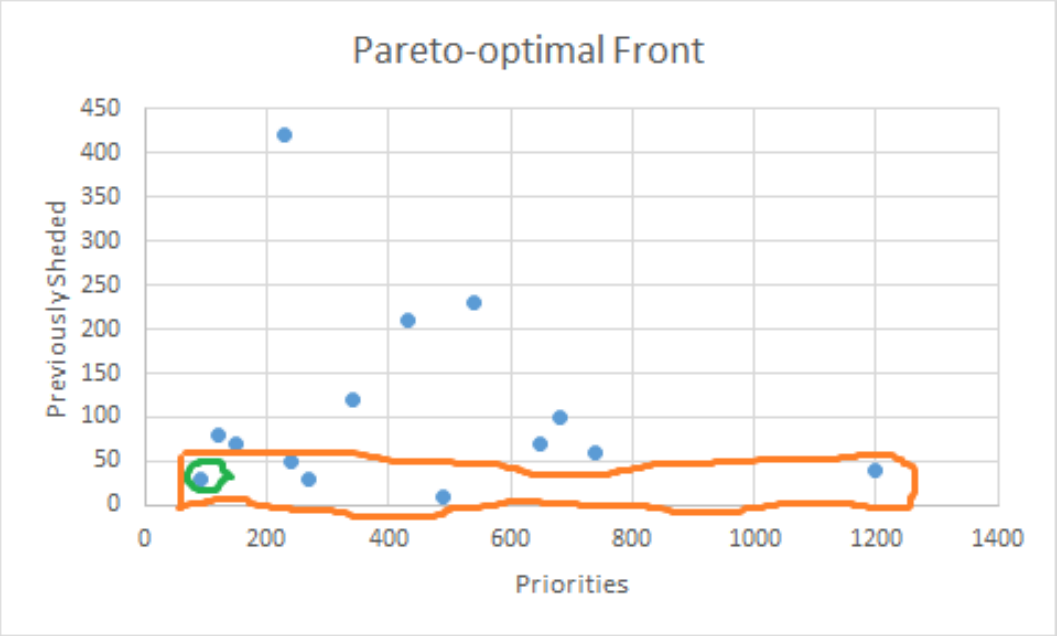

Figure 16 Example Pareto-optimal Front

The reason behind the usage of this methodology was to prioritize the previously shed objective as it is the objective with the highest priority for a fairer shedding strategy, and to make sure while prioritizing that objective we still avoid areas which have a high priority as much as possible.

4.3.2 Encoding

The encoding used for the solution in the genetic algorithm was a vector of doubles where a value greater than zero represents that an area is currently not shed, with a value of that double as the load of the area and can be used to evaluate the load of the solution as the sum of the vector or zero to represent that the area is currently shed.

4.3.3 Parameters Used

The parameters used in the algorithm are the following:

- Population Size: a population size of 1500 individuals was used in the algorithm is it was found to be a solution that resolves fast while providing considerably efficient solutions.
- Generations: the number of generations chosen was 200 generations as it was found to be a solution that resolves fast while providing considerably efficient solutions.
- Selection: Selection is applied by choosing 10 random individuals and then a tournament to find the most efficient individual from the selected ten is run using the crowding operator.
- Crossover: The crossover is applied once to create a child individual using two randomly selected parents from the population using the selection process described above. The single point crossover was the used method for applying crossover.
- Mutation: The percentage of genes mutated is 3% for a single mutation. It is applied once when an individual is created, when an individual is a duplicate it

is applied as much as necessary to make sure a unique individual is generated to explore the problem space it is also applied when an individual load compared to the original load is less than the load to be shed to make sure a solution is acceptable.

4.4 User Interface

The user interface was built using web technologies (HTML and JavaScript). D3 was used as it is a very known and comprehensive visualization library for JavaScript and is well documented with multiple templates.

After thorough research the multi-parent tree template was used for the following reasons:

- It is easy to use.
- It had a panning functionality as well as collapsing nodes which is very important considering the size of the grid.

The colour coding used for the user interface is:

- Yellow, for on.
- Red, for off.
- Orange, for on and previously shed.
- Black, for off and previously shed.

The interface is generated through the JavaScript script in the dndtree.js which loads the data every 30 seconds from the file named “grid.json”. This file is generated from the simulation software and is updated every time a change has occurred in the grid model. The file is taken in as input and parsed into the tree structure shown in the figures below. Then using the data provided to check if the area has been shed before or is currently shed creates a node using the color coding discussed above.

The user interface keeps the tree structure that we used in the model and shows how it is all connected since the model is too large to show in one figure it will be divided into the figures below.

The figure shown below displays the main grid node and all the regions, showing their load and generation in this case all remaining nodes have been collapsed for clarity.

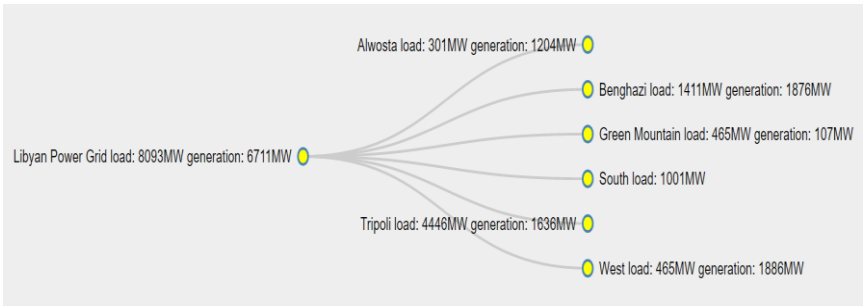

Figure 17 Grid Structure

The figure shown below shows the Alwosta region expanded and shows all areas and substations that belong to that region.

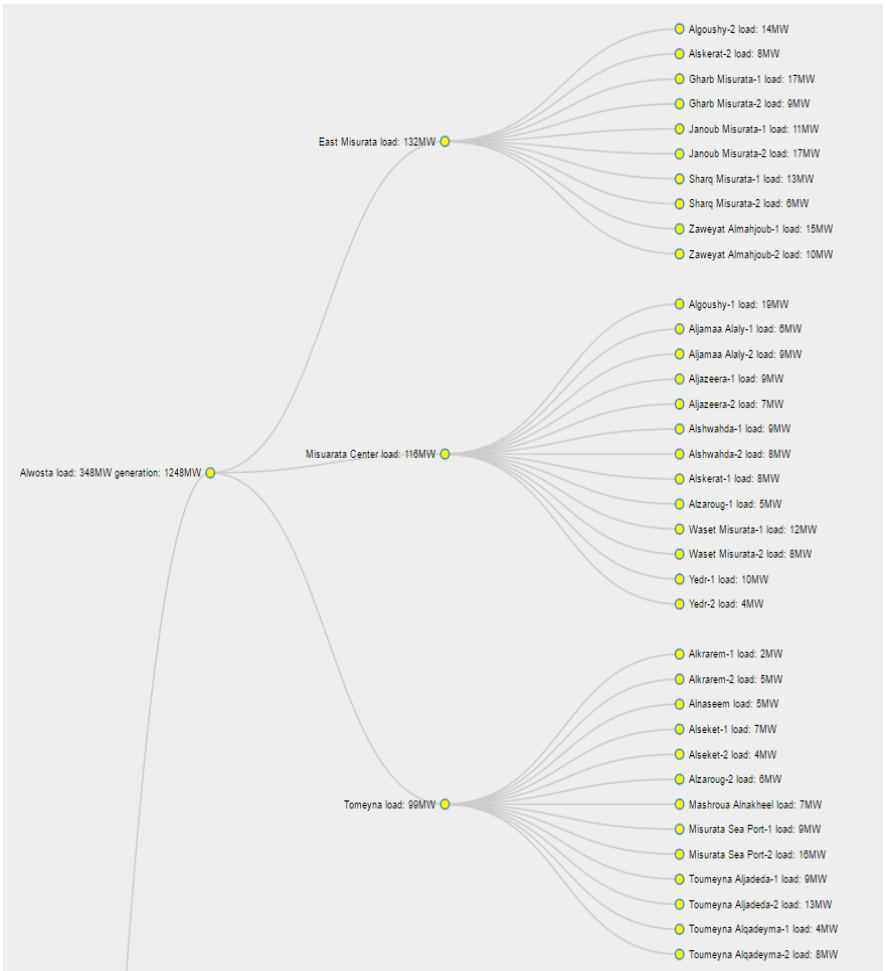

Figure 18 Alwosta Region

Finally, the figure shown below displays the changes that happened in the grid after load shedding has been applied, as can be seen the shedding is throughout the grid selecting areas from different regions which satisfies the goal of fairer load shedding between different geographical locations.

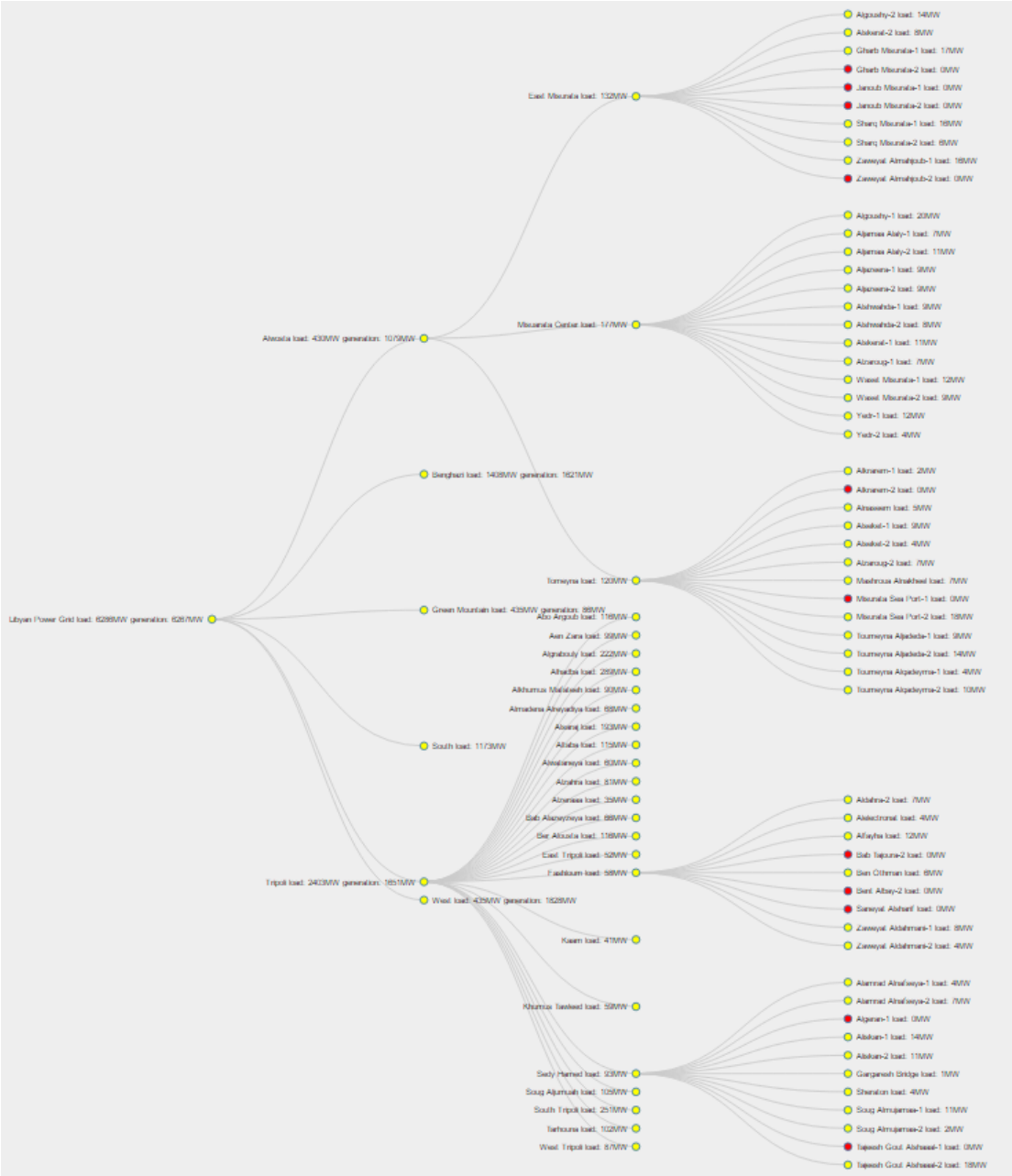

Figure 19 Example of Load Shedding

5 Evaluation

The evaluation process in this project can be divided into two sections, testing and analyzing.

5.1 Testing

Before we could start the analysis process of the implementation we first had to apply some tests to ensure that it behaves as expected.

The following tests were applied:

- Finding the most efficient fit was tested by running the simulation many times while changing the loads applied by areas and the amount of energy generated by the power plants. The environment was set so that the priorities were all equal and no areas were previously shed. This revealed that the solution that was chosen was the optimal solution based on the most efficient fit in the population of generated solutions.
- The avoidance of shedding the same area twice in a certain period will be tested by running the simulation many times. Such that the same area is the most applicable fit, then checking if the simulation decided to shed the same area twice in the chosen period. This yielded acceptable results however, increasing the number of generations yielded better results. The main reason for this is due to generating the individuals with a random value so it is very difficult to get optimal results unless the number of areas previously shed are small, a larger population size s used and is run for larger number of generations which would essentially slow the process of finding a solution.
- The avoidance of prioritized areas will be tested by running the simulation many times. Such that a prioritized area is the most appropriate fit to check if the simulation chooses that area to shed. This yielded great results on average where it completely avoided those areas.

As a result of the testing it can be seen that the implementation is behaving according to what is expected in each of the cases though in some areas it can be improved but due to the time constraints in the project it was not addressed.

5.2 Analysis

Since there is no known previous automated solution in the Libyan power grid and due to the time constraints this implementation could not be evaluated by comparing it to other existing solutions used in other studies. This will be evaluated based on how it chooses the optimal solution in a Pareto-front as seen in the examples below.

Best Fit	Previously Shed	Priorities
0.53	0.34	0.46
0.56	0.44	0.33
0.62	**0.45**	**0.26**
0.57	0.45	0.31
0.51	0.46	0.36
0.66	0.48	0.19
0.53	0.51	0.29
0.51	0.53	0.29
0.49	0.56	0.28
0.13	0.56	0.64
0.64	0.58	0.11

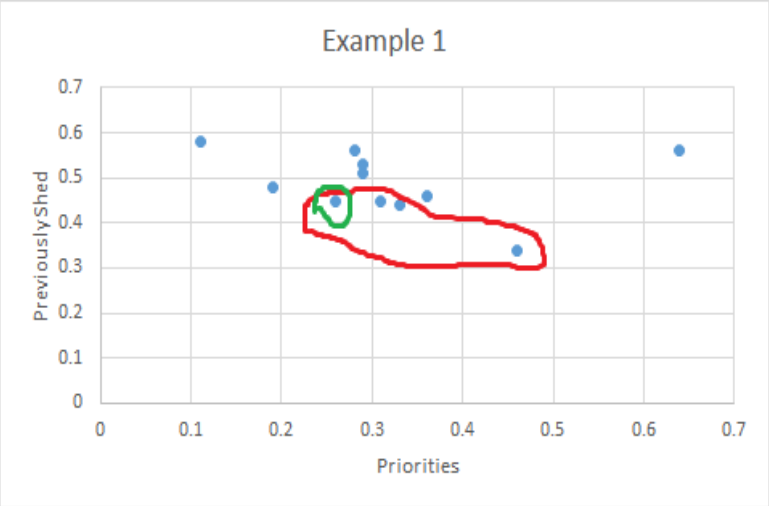

Figure 20 Example 1

In the first example shown above it can see that the solution chosen written in bold in the table is considered the best solution in this case as it has chosen a considerably small penalty for the previously shed objective and a more efficient candidate for the priorities objective.

In the example shown below it can be seen that there exist two solutions which are considered as better solutions for the following reasons:

- Blue Solution: a sacrifice of an increase of 0.01 in previously shed objective for decrease of 0.03 in the priorities objective.

- Purple Solution: a sacrifice of an increase of 0.02 in priorities objective for a decrease of 0.02 in the previously shed objective this encourages fairer shedding, which is a higher priority that justifies this choice over the others.

Best Fit	Previously Shed	Priorities
0.55	**0.41**	**0.31**
0.32	0.41	0.54
0.55	**0.43**	**0.29**
0.57	**0.44**	**0.26**
0.43	0.45	0.39
0.5	0.46	0.31
0.5	0.48	0.29
0.51	0.5	0.26
0.62	0.55	0.1
0.46	0.57	0.24
0.43	0.61	0.23

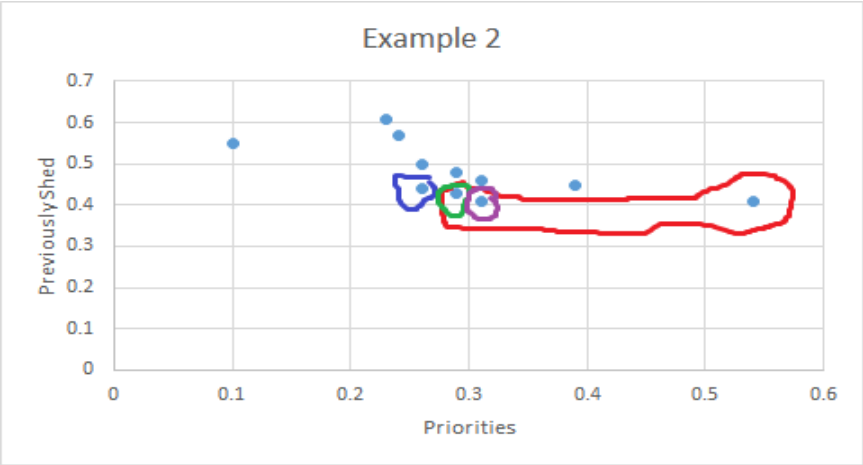

Figure 21 Example 2

In the example shown below it can be seen that the solution chosen can be considered as the optimal solution as it has chosen a solution with the most efficient tradeoffs compared to all other solutions in this front.

Best Fit	Previously Shed	Priorities
0.58	0.31	0.33
0.47	0.34	0.41
0.56	**0.43**	**0.23**
0.57	0.47	0.18
0.43	0.48	0.31
0.51	0.5	0.21
0.45	0.51	0.26
0.47	0.52	0.23
0.27	0.54	0.41
0.45	0.57	0.2
0.52	0.59	0.11

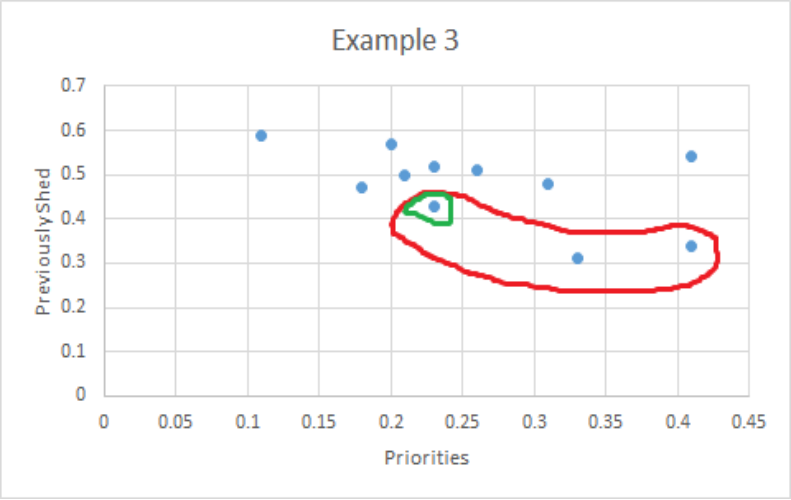

Figure 22 Example 3

In the final example seen below it can be observed that an optimal solution has been chosen, which also has the considerably most efficient tradeoffs between the available solutions.

Best Fit	Previously Shed	Priorities
0.47	0.13	0.55
0.42	0.22	0.51
0.38	0.26	0.51
0.45	0.34	0.36
0.29	0.35	0.51
0.4	0.4	0.35
0.41	0.42	0.32
0.5	0.45	0.2
0.12	0.45	0.58
0.34	0.45	0.36
0.2	0.59	0.36
0.21	0.59	0.35
0.14	0.66	0.35

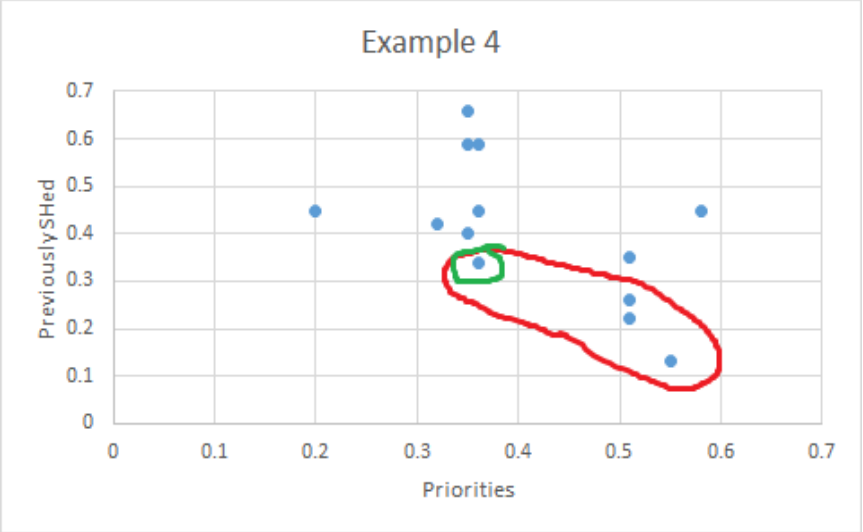

Figure 23 Example 4

The data discussed in the examples above was extracted from the final population's fronts to test how efficient is the implementation at choosing a solution that can be considered as the optimal solution in the selected front.

All the values for each objective have been normalized to ensure that they can be sorted into non dominant fronts with no preference of any objective as this will help in finding solutions that have the most efficient tradeoffs that satisfy the three objectives. As the numbers get lower it means that the solution will have a greater impact on the daily lives of the citizens of Libya by providing a fairer more efficient solution.

After running ten tests the average runtime found was about 2 minutes to find a solution using a population of 1500 and running for 200 generations, the results of those tests can be seen in the table below.

Test	load to be shed	load shed	prioritised areas	Medical areas	duplicate shedding
1	442.538	461.5	11	0	0
2	87.54	93.7	0	0	0
3	483.476	501.22	15	0	2
4	1012.895	1074.98	29	1	33
5	499.89	539.2	9	0	0
6	226.183	261	5	0	0
7	641.4949	687.78	17	0	3
8	333.801	354.9	7	0	0
9	280.362	297.7	6	0	0
10	451.703	487.48	11	0	11

From the results above we can see that as the load to be shed increases the other objectives are affected by this and the algorithm produces less efficient results, this can mainly be tackled by increasing the number of generations and the population size. Another possible method to tackle this issue is to optimize the selection function.

It can be seen that the algorithm is encouraging fairer load shedding as it tries to avoid the solutions with a larger number of areas that have been previously shed as much as possible. Though in the case were shedding has already been applied to a large amount of areas it becomes difficult to reduce the number of duplicates which is also due to the random algorithm used, which can be improved to find better results.

Also it can be observed that the algorithm tries to avoid the medical areas as much as it can. However, there are cases where medical areas are shed but since the implementation forces a maximum of two hours it should not be an issue as these areas are equipped with generators that will turn on automatically which will reduce the risk of life loss.

From what can be concluded from the previous examples, the implementation used in this project yields considerable optimal results in most cases, though there are times where it chooses a solution which isn't considered as the optimal solution as seen in example 2 and test 4 however, this is not the most occurring scenario from what has been observed while running the simulation.

Other factors that could be improved is the runtime speed for the shedding process, this can be improved using parallel paradigms and also by using a more efficient pseudo random number generator to reduce the creation of duplicate solutions which consumed a lot of time to ensure they are all removed.

5.3 Requirements Checklist

5.3.1 Functional Requirements

Non-Duplicate Shedding **Priority: High** **Achieved**

The provided implementation achieves non-duplicate as much as it possible can as in most cases it chooses a solution which has non-duplicate shedding as its highest priorities.

Prioritize Areas **Priority: Medium** **Achieved**

The provided implementation has taken prioritized shedding into consideration and is evident in the results.

Best Fit Shedding **Priority: Medium** **Achieved**

The provided implementation always chooses that best fit when appropriate though in some cases it will prioritize other objectives.

Visual Display of Simulation **Priority: Optional** **Achieved**

The provided implementation has a visual display included which can be accessed using a browser.

Day Ahead Prediction **Priority: Optional** **Not Achieved**

Due to time constraints this optional objective was not achieved but can be considered in the future.

5.3.2 Non-Functional Requirements

Cross Platform **Priority: Medium** **Achieved**

The provided implementation has been tested and runs with no observed issues on both platforms.

Framework Usage **Priority: Medium** **Not Achieved**

The usage of a framework was not achieved due to the complexity of the frameworks tested, the insufficient documentation provided with them and the time constraints on the project. However, an implementation of the algorithm required was completed which helped in achieving the completion of this project.

5.4 Objectives Checklist

Develop an Accurate Simulation **Achieved**

Based on the data provided by GECOL a near accurate simulation was implemented by discussing how to simulate some of the non-provided data e.g. (region loads).

2 Hour Shedding **Achieved**

This was achieved as the implementation unshed all areas after being shed for two hours.

Fair Shedding **Achieved**

Please refer to Section 5.3.1 Non-Duplicate Shedding which discusses this objective.

High Priority Remain Supplied **Achieved**

All solutions that include hospitals are given a high penalty compared to other solutions that can include other areas with lower priority to encourage the algorithm to avoid those areas.

6 Conclusion

In this chapter a discussion of the summary of the project, some of the limitations, possible future works that can be carried out to expand this project and finally a self-reflection will also be included.

6.1 Summary

This project was carried out as case study to showcase the usage of multi-objective genetic algorithms specifically the NSGA-II in order to simulate the automation of load shedding with the usage of the Libyan Power Grid as the underlying model it was operating on using C++ as the implementation language. The aim of this project was achieved to the best of the data available and time constraints, fairer load shedding than existing strategy, reduced shedding period to just 2 hours' maximum, prioritized areas and reduced duplicate shedding.

6.2 Limitations

6.2.1 Missing Areas

The main limitation of this implementation is that it does not include all the areas in the Libyan power grid, this is due to the time constraints placed on this project.

6.2.2 Realistic Data

Another limitation of this implementation is that the data is limited, which results in a less accurate simulation of the Libyan power grid.

6.3 Future Work

6.3.1 Scaling up

The next stage would be to scale up this implementation to include the whole Libyan power grid (all the areas outside the two supplied cities) and carry out further tests to achieve the highest accuracy possible.

6.3.2 Load Prediction

Another possible approach that could be carried out later on is to create a load predictor as discussed in section 2.4.1 and link it with the genetic algorithm. This would increase the ability to find better results as it gives the ability to run the shedding process for a greater number of generations and with a larger population of solutions as it will no longer be as time critical.

6.3.3 Improved User Interface

With more time a better user interface can be created, this can include functionalities as viewing all the previously shed areas in the current cycle as well as the ability to zoom out and zoom in to achieve a complete view of the power grid. Another great functionality would be to visit the historical data to help analyse and improve in the future.

6.3.3 Real-time System

The final goal would be to make this system into a real-time system which would be deployed in the main control room for the Libyan power grid, which takes in input in real-time and is able to shed the areas in the real world as that would achieve the main objective behind this research which is to help reduce the suffering of the Libyan people.

6.4 Reflection

In essence, I managed to accomplish the main objectives of my dissertation due to the usage of biologically inspired approaches combined with the computational performance benefits and abstraction of C++. I consider this as an accomplishment for the following reasons:

- I learned the object-oriented programming language C++.
- I learned how to use D3 visualization library.

- I learned multi-objective genetic algorithms, their inner workings and how the optimize and choose solutions from Pareto-optimal fronts.
- I gained knowledge on how electrical grids operate.

What makes me most proud is that, beyond the fact that I achieved the final step towards my graduation, I implemented a structure modeled after a real life power grid infrastructure. What makes me even more proud is that this solution could potentially be realized to help reduce the suffering of the citizens in my home country Libya.

7 References

[1] Libya [online], 2015. [online]. *Land Statistics*. Available from: http://www.worldatlas.com/webimage/countrys/africa/libya/lylandst.htm [Accessed 16 Nov 2015].

[2] Dictionary.com [online], 2015. [online]. *Load Shedding*. Available from: http://dictionary.reference.com/browse/load-shedding [Accessed 16 Nov 2015].

[3] Definition of artificial intelligence in English: [online], 2015. [online]. *artificial intelligence*. Available from: http://www.oxforddictionaries.com/definition/english/artificial-intelligence [Accessed 16 Nov 2015].

[4] Chen, S.H., Jakeman, A.J., and Norton, J.P., 2008. Artificial Intelligence techniques: An introduction to their use for modelling environmental systems. *Mathematics and Computers in Simulation*, 78 (2-3), 379–400.

[5] Programming a Perceptron in Python [online], 2015. [online]. *blogdbrgnch Full Atom*. Available from: https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/ [Accessed 16 Nov 2015].

[6] Wikipedia [online], 2015. [online]. *Artificial Neural Network*. Available from: https://en.wikipedia.org/wiki/artificial_neural_network [Accessed 16 Nov 2015].

[7] Fuzzy logic [online], 2015. [online]. *Fuzzy logic*. Available from: https://www.mcgill.ca/dis/research/fuzzy [Accessed 16 Nov 2015].

[8] 2 The Multipopulation Genetic Algorithm [online], 2015. [online]. *Multipopulation GA*. Available from: http://www.pohlheim.com/papers/mpga_gal95/gal2_2.html [Accessed 16 Nov 2015].

[9] Tanenbaum, A.S., 1992. *Modern operating systems*. Englewood Cliffs, NJ: Prentice Hall.

[10] Jain, A., Srinivas, E., and Kukkadapu, S.K., 2010. Fuzzy based day ahead prediction of electric load using Mahalanobis distance. *2010 International Conference on Power System Technology*.

[11] Hsu, C.-T., Kang, M.-S., and Chen, C.-S., 2005. Design of adaptive load shedding by artificial neural networks. *IEE Proc., Gener. Transm. Distrib. IEE Proceedings - Generation, Transmission and Distribution*, 152 (3), 415.

[12] Rao, K., Ganeshprasad, G., Pillappa, S., Jayaprakash, G., and Bhat, S., 2013. Time priority based optimal load shedding using genetic algorithm. *Fifth International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom 2013)*.

[13] Department of Computer Science Georgia State University Atlanta, 2004. *Weighted Round-Robin CPU Scheduling Algorithm.*

[14] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation IEEE Trans. Evol. Computat.*, 6 (2), 182–197.

[15] Optimal Solutions of Multiproduct Batch Chemical Process Using Multiobjective Genetic Algorithm with Expert Decision System : Figure 1 [online], 2015. [online]. *Optimal Solutions of Multiproduct Batch Chemical Process Using Multiobjective Genetic Algorithm with Expert Decision System : Figure 1*. Available from: http://www.hindawi.com/journals/jamc/2009/927426/fig1/ [Accessed 16 Nov 2015].

[16] Corne, David W., et al., 2001. "PESA-II: Region-based selection in evolutionary multiobjective optimization." *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO* 2001).

[17] C programmers to benefit from studying the manuals [online], 2015. [online]. *ComputerWeekly*. Available from: http://www.computerweekly.com/feature/c-programmers-to-benefit-from-studying-the-manuals [Accessed 16 Nov 2015].

[18] Open BEAGLE [online], 2015. [online]. *beagle -: A Generic Evolutionary Computation Framework in C++.* Available from: https://code.google.com/p/beagle/ [Accessed 22 Nov 2015].

[19] Garg, R. and Singh, A.K., 2011. Multi-Objective Optimization to Workflow Grid Scheduling using Reference Point based Evolutionary Algorithm. *International Journal of Computer Applications IJCA*, 22 (6), 44–49.

[20] Wang, H., Fu, Y., and Huang, M., 2015. A species based multiobjective evolutionary algorithm for multiobjective flow shop scheduling problem. *2015 IEEE Congress on Evolutionary Computation (CEC)*.

8 Appendences

8.1 Simulation

8.1.1 Simulation.cpp

```
// Grid-Model.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "region.h"
#include "pugixml.hpp"
#include "nsga2.h"
#include <fstream>
#include <random>
#include <time.h>
#include <map>
std::vector<region> regions;
std::vector<powerStation> powerStations;
std::vector<subStation> subStations;
std::vector<area> areas;
std::map<int, int> previously_sheded;
std::vector<double> base;
grid g;
int medical_areas = 0;
int time_now = 8;


int load_grid()
{
	std::string source = "regions.xml";

	pugi::xml_document doc;
	if (!doc.load_file(source.c_str())) return -1;

	std::cout << "parsing regions" << std::endl;
	pugi::xml_node regions = doc.child("regions");

	for (pugi::xml_node reg = regions.first_child(); reg; reg =
reg.next_sibling())
	{
		region r;

		r.id = atoi(reg.child("id").child_value());
		r.name = reg.child("name").child_value();
		r.gridPointer = &::g;
		::regions.push_back(r);
	}
	for (int h = 0; h < ::regions.size(); h++)
	{
		g.regions.push_back(&::regions.at(h));
	}

	std::cout << "\nparsing substations" << std::endl;
	source = "substations.xml";

	if (!doc.load_file(source.c_str())) return -1;


	pugi::xml_node substations = doc.child("substations");
```

```
	for (pugi::xml_node substation = substations.first_child();
substation; substation = substation.next_sibling())
	{
		subStation s;

		s.id = atoi(substation.child("id").child_value());
		s.name = substation.child("name").child_value();
		int parent_id = atoi(substation.child("region").child_value());
		s.regionPointer = &::regions.at(parent_id - 1);
		::subStations.push_back(s);

	}
	for (int f = 0; f < ::subStations.size(); f++)
	{
		subStation s = ::subStations.at(f);
		g.regions.at(s.regionPointer->id - 1)-
>subStations.push_back(&::subStations.at(f));
	}

	std::cout << "\nparsing powerstations" << std::endl;
	source = "powerstations.xml";

	if (!doc.load_file(source.c_str())) return -1;


	pugi::xml_node powerstations = doc.child("powerstations");

	for (pugi::xml_node powerstation = powerstations.first_child();
powerstation; powerstation = powerstation.next_sibling())
	{
		powerStation p;

		p.id = atoi(powerstation.child("id").child_value());
		p.name = powerstation.child("name").child_value();
		int parent_id =
atoi(powerstation.child("region").child_value());
		p.regionPointer = &::regions.at(parent_id - 1);
		p.capacity =
atof(powerstation.child("capacity").child_value());
		::powerStations.push_back(p);

	}
	for (int f = 0; f < ::powerStations.size(); f++)
	{
		powerStation p = ::powerStations.at(f);
		g.regions.at(p.regionPointer->id - 1)-
>powerStations.push_back(&::powerStations.at(f));
	}

	std::cout << "\nparsing areas\n" << std::endl;
	source = "areas.xml";

	if (!doc.load_file(source.c_str())) return -1;


	pugi::xml_node areas = doc.child("areas");

	for (pugi::xml_node ar = areas.first_child(); ar; ar =
ar.next_sibling())
	{
		area a;
		a.id = atoi(ar.child("id").child_value());
		a.name = ar.child("name").child_value();
```

```
		a.area_type = ar.child("type").child_value();
		int parent_id = atoi(ar.child("substation").child_value());
		a.subStationPointer = &::subStations.at(parent_id - 1);
		::areas.push_back(a);
		if (a.area_type == "Medical")
			medical_areas++;

	}

	int count = 0;
	for (int d = 0; d < ::areas.size(); d++)
	{
		area a = ::areas.at(d);
		int parent_id = a.subStationPointer->id;
		for (int i = 0; i < g.regions.at(a.subStationPointer-
>regionPointer->id - 1)->subStations.size(); i++)
		{
			if (parent_id == g.regions.at(a.subStationPointer-
>regionPointer->id - 1)->subStations.at(i)->id)
			{
				count++;

				g.regions.at(a.subStationPointer->regionPointer-
>id - 1)->subStations.at(i)->areas.push_back(&::areas.at(d));

				break;
			}
		}
	}
	std::cout << "Number of areas: " << count << std::endl;

}


void init_grid()
{
	// initialize the loads
	std::ifstream stream("loads.txt");
	std::string line = "";
	for (int l = 0; l < ::areas.size(); l++)
	{
		std::getline(stream, line);
		::areas.at(l).init(atof(line.c_str()));
	}
	stream.close();

	// initialize the generation
	for (int i = 0; i < ::powerStations.size(); i++)
	{
		powerStations.at(i).init(powerStations.at(i).capacity * .8);
	}
	regions.at(3).generation = 0;
	regions.at(2).update(500, 0);
	regions.at(3).update(1000, 0);
	regions.at(4).update(1500, 0);
	regions.at(5).update(500, 0);
	for (int i = 0; i < ::areas.size(); i++)
	{
		::base.push_back(areas.at(i).load);
	}

}

void update()
```

```
{
	for (int i = 0; i < ::areas.size(); i++)
	{
		::areas.at(i).update();
	}
	for (int i = 0; i < ::powerStations.size(); i++)
	{
		powerStations.at(i).update();
	}
	for (int i = 2; i < 6; i++)
	{
		double l = 0;
		srand(time(NULL));
		l = -(regions.at(i).load * .1) + (fmod(std::rand(),
(regions.at(i).load * .1) * 2));
		regions.at(i).update(l, 0);
	}
}

void wait(int seconds)
{
	clock_t endwait;
	endwait = clock() + seconds * CLOCKS_PER_SEC;
	while (clock() < endwait) {}
}
void shed()
{
	std::cout << "shedding" << std::endl;
	std::vector<double> grid;
	for (int i = 0; i < ::areas.size(); i++)
	{
		grid.push_back(::areas.at(i).load);
	}
	std::vector<area> ps;
	typedef std::map<int, int>::iterator it_type;
	for (it_type iterator = previously_sheded.begin(); iterator !=
previously_sheded.end(); iterator++)
	{
		ps.push_back(::areas.at(iterator->first));
	}
	NSGA2 nsga2(true, 10, 10, 200, 1500, ::areas, ps, base, grid,
abs((g.generation - g.load) - (g.generation * .1)), time_now);
	std::vector<int> res = nsga2.evolve();
	double amount = 0;
	int residential = 0;
	int economic = 0;
	int governmental = 0;
	int medical = 0;
	int sheded = 0;
	for (int i = 0; i < res.size(); i++)
	{
		if (::areas.at(res.at(i)).load != 0)
		{
			amount += ::areas.at(res.at(i)).load;
			::areas.at(res.at(i)).apply_shedding(0);
			if (::areas.at(res.at(i)).area_type == "Residential")
				residential++;
			if (::areas.at(res.at(i)).area_type == "Economical")
				economic++;
			if (::areas.at(res.at(i)).area_type == "Governmental")
				governmental++;
			if (::areas.at(res.at(i)).area_type == "Medical")
				medical++;
```

```
                        try {
                                previously_sheded[res.at(i)]++;
                                if (previously_sheded[res.at(i)] > 3)
                                        sheded++;
                        }
                        catch (const std::exception& e)
                        {
                                previously_sheded.insert(std::pair<int,
int>(res.at(i), 1));
                        }
                }
        }
        std::cout << "sheded size " << previously_sheded.size() << "\nmedical
areas in grid: " << medical_areas << std::endl;
        std::cout << "number of areas sheded " << res.size() << std::endl;
        std::cout << "duplicate shedding: " << sheded << std::endl;
        std::cout << "prioritised areas sheded: " << medical + governmental +
economic << "\nmedical: " << medical << "\tgovernmental: " << governmental <<
"\teconomical: " << economic << "\tresidential: " << residential <<
std::endl;

        if (previously_sheded.size() > ::areas.size() - medical_areas)
        {
                previously_sheded.clear();
                std::cout << "\nworked\n" << std::endl;
        }
        std::cout << "load sheded " << amount << std::endl;
}

void unshed(std::vector<int> a)
{
        for (int i = 0; i < a.size(); i++)
        {
                ::areas.at(i).apply_shedding(base.at(a.at(i)));
        }
}

int main()
{

        if (load_grid() == -1) return -1;

        init_grid();

        g.print(previously_sheded);
        for (int i = 0; i < g.regions.size(); i++)
        {
                g.regions.at(i)->print();

        }
        std::vector<int> areas_to_unshed;
        while (true)
        {
                areas_to_unshed.clear();
                typedef std::map<int, int>::iterator it_type;
                for (it_type iterator = previously_sheded.begin(); iterator !=
previously_sheded.end(); iterator++)
                {
                        iterator->second++;
                        if (iterator->second > 2)
                                areas_to_unshed.push_back(iterator->first);
                }
                unshed(areas_to_unshed);
                update();
```

```
		if (time_now < 10)
			std::cout << "time " << "0" << time_now << ":00" <<
std::endl;
		else
			std::cout << "time " << time_now << ":00" << std::endl;
		//g.generation = 6000;
		g.print(previously_sheded);

		if (g.generation - g.load < g.generation * .10)
		{
			shed();
			if (time_now < 10)
				std::cout << "time " << "0" << time_now << ":00"
<< std::endl;
			else
				std::cout << "time " << time_now << ":00" <<
std::endl;
			g.print(previously_sheded);
		}


		time_now++;
		if (time_now > 23)
			time_now = 0;

		int id = 0;
		wait(30);
	}
	return 0;
}
```

8.1.2 Stdafx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

//#include "targetver.h"

#include <stdio.h>
//#include <tchar.h>
#include <string>
#include <vector>
#include <iostream>



// TODO: reference additional headers your program requires here
```

8.2 Grid-Model

8.2.1 Grid.h

```
#pragma once
#include "stdafx.h"
#include "region.h"
#include "area.h"
#include <map>
class region;
class area;
class grid {
public:
	std::string name;// = "Libyan Power Grid";
	double generation;
	double load;
	std::vector<region*> regions;
	void update(double l, double g);
	void shed();
	void unshed();
	void change();
	void print(std::map<int, int> previously_sheded);

};
```

8.2.2 Grid.cpp

```
#include "stdafx.h"
#include "grid.h"
#include "nsga2.h"
#include <fstream>
void grid::change() {

}

void grid::shed() {

}

void grid::unshed() {

}

void grid::print(std::map<int, int> previously_sheded) {



	std::cout << "grid name: " << "Libyan Power Grid" << "\n" <<
"threshold: " << generation - load << " / " << generation * .1
		<< "\n" << "grid generation: " << generation << "\n" << "grid
load: " << load << "\n" << std::endl;
	std::string result = "";//"[\n\t";
	result += "{\n\t\"name\":\"Libyan Power Grid\",\n\t\"load\":" +
std::to_string(static_cast<long long>(load)) +
		",\n\t\"generation\":" + std::to_string(static_cast<long
long>(generation)) + ",\n\t" +
		"\"previously sheded\": [\n";
		int index = 0;
		typedef std::map<int, int>::iterator it_type;
```

```
            for(it_type iterator = previously_sheded.begin(); iterator !=
previously_sheded.end(); iterator++)
            {
                  if(index < previously_sheded.size() - 1)
                  {
                        result += "\t{\n\t\"id\":" +
std::to_string(static_cast<long long>(iterator->first)) + ",\n" +
                                          "\t\"sheded\":" +
std::to_string(static_cast<long long>(iterator->second)) + "\n},\n";
                  }
                  else{
                        result += "\t{\n\t\"id\":" +
std::to_string(static_cast<long long>(iterator->first)) + ",\n" +
                                          "\t\"sheded\":" +
std::to_string(static_cast<long long>(iterator->second)) + "\n}\n";
                  }
                  index++;
            }
                        result +=  "],\n\"children\": [\n";
      for (int i = 0; i < regions.size(); i++)
      {
            result += "\t\t{\n\t\t\"name\": \"" + regions.at(i)->name +
"\"" +
                        ",\n\t\t\"load\":" +
std::to_string(static_cast<long long>(regions.at(i)->load)) +
                  ",\n\t\t\"generation\":" +
std::to_string(static_cast<long long>(regions.at(i)->generation)) + ",\n\t" +
                     "\t\"children\": [\n";
            for (int j = 0; j < regions.at(i)->subStations.size(); j++)
            {
                  result += "\t\t\t{\n\t\"name\": \"" + regions.at(i)-
>subStations.at(j)->name + "\"" +
                        ",\n\t\t\t\"load\":" +
std::to_string(static_cast<long long>(regions.at(i)->subStations.at(j)-
>load)) +
                   ",\n\t\t\t\"children\": [\n";
                  for (int k = 0; k < regions.at(i)->subStations.at(j)-
>areas.size(); k++)
                  {
                        int sheded = 0;
                        typedef std::map<int, int>::iterator it_type;
                        for (it_type iterator =
previously_sheded.begin(); iterator != previously_sheded.end(); iterator++)
                        {
                              //std::cout << "second " << iterator-
>second << std::endl;
                              if (iterator->first == regions.at(i)-
>subStations.at(j)->areas.at(k)->id && iterator->second > 2)
                                    sheded = 1;
                        }
                        result += "\t{\n\t\t\t\t\"name\": \"" +
regions.at(i)->subStations.at(j)->areas.at(k)->name + "\"" +
                              ",\n\t\t\t\t\"load\":" +
std::to_string(static_cast<long long>(regions.at(i)->subStations.at(j)-
>areas.at(k)->load)) +
                              ",\n\t\t\t\t\"sheded\":" +
std::to_string(static_cast<long long>(sheded));
                        if (k < regions.at(i)->subStations.at(j)-
>areas.size() - 1)
                              result += "\n\t\t\t},\n\t\t\t";
                        else
                              result += "\n\t\t\t}\n\t\t\t";
```

```
                        //std::cout << k << "jaofj\n" << result <<
std::endl;
                }
                if (j < regions.at(i)->subStations.size() - 1)
                        result += "\n\t]\n\t},\n";
                else
                        result += "\n\t]\n\t}\n";
        }
        if (i < regions.size() - 1)
                result += "\n\t]\n\t},\n";
        else
                result += "\n\t]\n\t}\n";
    }

    result += "\n\t]\n\t}";//\n]";
    std::ofstream myfile;
    myfile.open("grid.json");
    myfile << result;
    myfile.close();

}

void grid::update(double l, double g) {
    load += l;
    generation += g;
    if (generation - load < (generation * .1))
    {
        shed();
    }
    else if (generation - load > (generation * .15))
    {
        unshed();
    }
}
```

8.2.3 Region.h

```
#pragma once
#include "grid.h"
#include "powerStation.h"
#include "subStation.h"
class grid;
class powerStation;
class subStation;
class region {
public:
    int id;
    std::string name;
    double generation;
    double load;
    std::vector<powerStation*> powerStations;
    std::vector<subStation*> subStations;
    void update(double l, double g);
    void print();
    grid * gridPointer;
};
```

8.2.4 Region.cpp

```
#include "stdafx.h"
#include "region.h"


void region::update(double l, double g)
{
	load += l;
	generation += g;
	gridPointer->update(l, g);
}

void region::print() {
	std::cout << "region id: " << id << "\n" << "region name: " << name <<
"\n" << "region generation: " << generation << "\n" << "region load: " <<
load << "\n" << std::endl;
}
```

8.2.5 Powerstation.h

```
#pragma once
#include "region.h"
class region;
class powerStation {
public:
	int id;
	std::string name;
	double generation;// = 0;
	double previousGeneration;
	double capacity;
	void init(double g);
	void update();
	void print();
	region* regionPointer;
};
```

8.2.6 Powerstation.cpp

```
#include "stdafx.h"
#include "powerStation.h"
#include <random>
#include <time.h>

void powerStation::init(double g)
{
	previousGeneration = 0;
	generation = g;
	regionPointer->update(0, generation - previousGeneration);
}

void powerStation::update()
{
	double g = 0;
	if(generation != 0)
```

```
	{
		srand(time(NULL) * 3);
		g = -(generation * .1) + (fmod(std::rand(), (generation * .1) *
2));
		previousGeneration = generation;
		generation += g;
		if(generation < capacity * .6)
		{
			generation = capacity * .6;
			g = generation - previousGeneration;
		}else if(generation > capacity * .9)
		{
			generation = capacity * .9;
			g = generation - previousGeneration;
		}
		regionPointer->update(0, g);

	}
}

void powerStation::print() {
	std::cout << "power station id: " << id << "\n" << "power station
name: " << name << "\n" << "power station generation: " << generation
		<< "\n" << "power station capacity: " << capacity << "\n" <<
std::endl;
}
```

8.2.7 Substation.h

```
#pragma once
#include "region.h"
#include "area.h"
class area;
class region;
class subStation {
public:
	int id;
	std::string name;
	double load;
	std::vector<area*> areas;
	void update(double l);
	void print();
	region * regionPointer;
};
```

8.2.8 Substation.cpp

```
#include "stdafx.h"
#include "subStation.h"


void subStation::update(double l)
{
	load += l;
	regionPointer->update(l, 0);
}
```

```
void subStation::print() {
	std::cout << "sub station id: " << id << "\n" << "sub station name: "
<< name << "\n" << "sub station load: " << load << "\n" << std::endl;
}
```

8.2.9 Area.h

```
#pragma once
#include "region.h"
#include "area.h"
class subStation;
class area {
public:
	int id;
	std::string name;
	std::string area_type;
	double previousLoad;
	double load;
	double baseLoad;
	subStation * subStationPointer;
	void update();
	void init(double l);
	void print();
	void apply_shedding(double l);
};
```

8.2.10 Area.cpp

```
#include "stdafx.h"
#include "area.h"
#include <random>
#include <time.h>

void area::init(double l)
{
	previousLoad = 0;
	load = l;
	baseLoad = l;
	subStationPointer->update(load);
}

void area::update()
{
	int l = 0;
	if (load != 0)
	{
		srand(time(NULL) * std::rand() * 5);
		l = -(load * .1) + fmod(std::rand(), (load * .1) * 2);
		previousLoad = load;
		load += l;
		if(load < baseLoad * .8)
		{
			load = baseLoad * .8;
			l = load - previousLoad;
		}else if(load > baseLoad * 1.4)
		{
			load = baseLoad * 1.4;
			l = load - baseLoad;
```

```
                }
                subStationPointer->update(l);
        }
}

void area::apply_shedding(double l)
{
        previousLoad = load;
        load = l;
        subStationPointer->update(load - previousLoad);
}

void area::print() {
        std::cout << "area id: " << id << "\n" << "area name: " << name <<
"\n" << "substation name: " << subStationPointer->name << "\n" << "area load:
" << load << std::endl;
}
```

8.3 NSGA-II

Below is the code used in the implementation of the NSGA-II algorithm and all related functions, which is used in the evolution of the solutions.

8.3.1 Individual.h

```
#pragma once
#include "stdafx.h"
class Individual {
public:
	//variables
	int rank;
	double crowding_distance;
	std::vector<double> cd;
	int domination_count;
	std::vector<Individual*> dominated_solutions;
	std::vector<double> features;
	std::vector<double> objectives;

	//functions
	Individual();
	bool dominates(Individual* o);
};
```

8.3.2 Individual.cpp

```
#include "stdafx.h"
#include "individual.h"
Individual::Individual()
{
	cd = { 0,0,0 };
}
bool Individual::dominates(Individual * o)
{
	int result1 = 0;
	int result2 = 0;

	for (int i = 0; i < 3; i++)
	{
		result1 += objectives.at(i);
		result2 += o->objectives.at(i);
	}

	if (result1 <= result2 && medical == 0)
		return true;
	else
		return false;

}
```

8.3.3 Population.h

```
#pragma once
#include "stdafx.h"
#include "individual.h"
class Population {
public:
	std::vector<Individual> population;
	std::vector<std::vector<Individual>> fronts;
};
```

8.3.4 Population.cpp

```
#include "stdafx.h"
#include "powerStation.h"
#include <random>
#include <time.h>

void powerStation::init(double g)
{
	previousGeneration = 0;
	generation = g;
	regionPointer->update(0, generation - previousGeneration);
}

void powerStation::update()
{
	double g = 0;
	if(generation != 0)
	{
		srand(time(NULL) * 3);
		g = -(generation * .1) + (fmod(std::rand(), (generation * .1) *
2));
		previousGeneration = generation;
		generation += g;
		if(generation < capacity * .6)
		{
			generation = capacity * .6;
			g = previousGeneration - generation;
		}else if(generation > capacity * .9)
		{
			generation = capacity * .9;
			g = generation - previousGeneration;
		}
		regionPointer->update(0, g);

	}
}

void powerStation::print() {
	std::cout << "power station id: " << id << "\n" << "power station
name: " << name << "\n" << "power station generation: " << generation
		<< "\n" << "power station capacity: " << capacity << "\n" <<
std::endl;
}
```

8.3.5 NSGA2.h

```
#pragma once
#include "stdafx.h"
#include "population.h"
#include <numeric>
#include "area.h"

# define INF 1.0e14

class NSGA2 {
public:
	//variables
	double times;
	std::vector<area> areas;
	std::vector<area> previously_sheded;
	std::vector<double> base;
	std::vector<double> grid;

	double load;
	double load_to_be_shed;
	int num_of_individuals;// = 0;
	double mutation_strength;
	int number_of_genes_to_mutate;// = 0;
	int num_of_tour_participants;// =2;
	int num_of_generations;
	int num_of_participants;
	Population population;
	bool shed;// = true;

	//functions
	void BubbleSort(std::vector<Individual>& num);
	Individual chooseOptimal(std::vector<Individual>& num);
	NSGA2(bool shed, int num_of_individual, int num_of_genes_to_mutate,
int num_of_generations, int num_of_participants, std::vector<area> areas,
std::vector<area> previously_sheded, std::vector<double> base,
std::vector<double> grid, double load_to_be_shed, double time);
	std::vector<int> evolve();
	void fast_nondominated_sort(Population& p);
	std::vector<Individual>
calculate_crowding_distance(std::vector<Individual>& front);
	bool crowding_operator(Individual i, Individual j);
	Individual generate_individual();
	void calculate_objectives(Individual & individual);
	void create_initial_population();
	std::vector<Individual> create_children();
	std::vector<Individual> crossover(Individual a, Individual b);
	void mutate(Individual & child);
	Individual tournament();
};
```

8.3.6 NSGA2.cpp

```
#include "stdafx.h"
#include "nsga2.h"
#include <time.h>
#include <random>
#include <algorithm>
```

```
#include <fstream>
#include <math.h>

NSGA2::NSGA2(bool s, int individual, int genes_to_mutate, int generations,
int num_participants, std::vector<area> a, std::vector<area> sheded,
std::vector<double> b, std::vector<double> g, double ltbs, double t)
{

	number_of_genes_to_mutate = genes_to_mutate;
	num_of_individuals = individual;
	shed = s;
	num_of_generations = generations;
	shed = true;
	num_of_tour_participants = num_participants;
	grid = g;
	previously_sheded = sheded;
	base = b;
	areas = a;
	load_to_be_shed = ltbs;
	load = std::accumulate(grid.begin(), grid.end(), 0.0);
	times = t;
}

std::vector<int> NSGA2::evolve()
{
	srand(time(NULL));
	Population result;
	create_initial_population();
	fast_nondominated_sort(population);
	for (int i = 0; i < population.fronts.size(); i++)
	{
		calculate_crowding_distance(population.fronts.at(i));
	}
	std::vector<Individual> children = create_children();
	for (int j = 0; j < num_of_generations; j++)
	{
		population.population.insert(population.population.end(),
children.begin(), children.end());
		fast_nondominated_sort(population);
		Population new_population;
		int front_num = 0;
		while (new_population.population.size() +
population.fronts.at(front_num).size() < num_of_individuals)
		{

	calculate_crowding_distance(population.fronts.at(front_num));

	new_population.population.insert(new_population.population.end(),
population.fronts.at(front_num).begin(),
population.fronts.at(front_num).end());
			front_num++;
		}
		NSGA2::BubbleSort(population.fronts.at(front_num));
		for (int k = 0; new_population.population.size() <
num_of_individuals; k++)
		{

	new_population.population.push_back(population.fronts.at(front_num).at
(k));
		}
		result = population;
		population = new_population;
		children = create_children();
	}
```

```
	Individual best = chooseOptimal(result.fronts.at(0));
	std::vector<int> final_result;
	for (int k = 0; k < best.features.size(); k++)
	{
		if (best.features.at(k) == 0)
			final_result.push_back(k);
	}
	return final_result;
}

void NSGA2::BubbleSort(std::vector<Individual>& num)
{
	int i, j, flag = 1;    // set flag to 1 to start first pass
	Individual temp;             // holding variable
	int numLength = num.size();
	for (i = 1; (i <= numLength) && flag; i++)
	{
		flag = 0;
		for (j = 0; j < (numLength - 1); j++)
		{
			if (crowding_operator(num[j + 1], num[j]))      //
ascending order simply changes to <
			{
				temp = num[j];             // swap elements
				num[j] = num[j + 1];
				num[j + 1] = temp;
				flag = 1;               // indicates that a swap
occurred.
			}
		}
	}
	//arrays are passed to functions by address; nothing is returned
}

Individual NSGA2::chooseOptimal(std::vector<Individual>& num)
{
	int i, j, flag = 1;    // set flag to 1 to start first pass
	Individual temp;             // holding variable
	int numLength = num.size();
	for (i = 1; (i <= numLength) && flag; i++)
	{
		flag = 0;
		for (j = 0; j < (numLength - 1); j++)
		{
			if (num[j + 1].objectives.at(1) <
num[j].objectives.at(1))      // ascending order simply changes to <
			{
				temp = num[j];             // swap elements
				num[j] = num[j + 1];
				num[j + 1] = temp;
				flag = 1;               // indicates that a swap
occurred.
			}
		}
	}
	int len;
	if (numLength > 3)
		len = numLength / 3;
	else
		len = numLength;
	int index = -1;
	int best = -1;
	for (int b = 0; b < len; b++)
	{
```

```
		if (best == -1 || best < num.at(b).objectives.at(2))
		{
			best = num.at(b).objectives.at(2);
			index = b;
		}
	}
	return num.at(index);
}


void NSGA2::fast_nondominated_sort(Population& p)
{
	for (int m = 0; m < p.population.size(); m++)
	{
		p.population.at(m).domination_count = 0;
		calculate_objectives(p.population.at(m));
	}
	for (int m = 0; m < 3; m++)
	{
		double best = 1000000000000;
		double worst = -10000000000;
		for (int i = 0; i < p.population.size(); i++)
		{
			if (p.population.at(i).objectives.at(m) < best)
				best = p.population.at(i).objectives.at(m);
			if (p.population.at(i).objectives.at(m) > worst)
				worst = p.population.at(i).objectives.at(m);

		}
		if (worst == 0)
			worst = 1;
		for (int j = 0; j < p.population.size(); j++)
		{
			p.population.at(j).objectives.at(m) =
(p.population.at(j).objectives.at(m) - best) / (worst - best);
			p.population.at(j).objectives.at(m) =
ceil((p.population.at(j).objectives.at(m) * pow(10, 2)) - 0.49) / pow(10, 2);
		}
	}
	std::vector<Individual> temp;
	for (int i = 0; i < p.population.size(); i++)
	{
		for (int j = 0; j < p.population.size(); j++)
		{
			if (i != j)
			{
				if
(p.population.at(i).dominates(&p.population.at(j)))

	p.population.at(i).dominated_solutions.push_back(&p.population.at(j));
				else
					p.population.at(i).domination_count++;
			}
		}
		if (p.population.at(i).domination_count == 0)
		{
			p.population.at(i).rank = 1;
			temp.push_back(p.population.at(i));
		}
	}
	int front;
	if (temp.size() > 0)
	{
		p.fronts.push_back(temp);
```

```
                front = 1;
        }
        else {
                front = 0;
        }
        while (true)
        {
                bool b = true;
                std::vector<Individual> t;
                for (int k = 0; k < p.population.size(); k++)
                {
                        p.population.at(k).domination_count--;
                        if (p.population.at(k).domination_count == 0)
                        {
                                p.population.at(k).rank = front + 1;
                                t.push_back(p.population.at(k));
                        }
                        if (p.population.at(k).domination_count > 0)
                                b = false;
                }
                if (t.size() > 0)
                {
                        p.fronts.push_back(t);
                        front++;
                }
                if (b)
                {
                        break;
                }
        }
}

std::vector<Individual>
NSGA2::calculate_crowding_distance(std::vector<Individual>& front)
{
        if (front.size() > 0)
        {
                std::vector<Individual> temp;
                int solutions_size = front.size();
                std::vector<int> inserted;
                double best = 1000000;
                int index = 1000000;
                for (int i = 0; i < 3; i++)
                {

                        int k, j, flag = 1;    // set flag to 1 to start first
pass
                        Individual temp;             // holding variable
                        int numLength = front.size();
                        for (k = 1; (k <= numLength) && flag; k++)
                        {
                                flag = 0;
                                for (j = 0; j < (numLength - 1); j++)
                                {
                                        if (front[j + 1].objectives.at(i) <
front[j].objectives.at(i))      // ascending order simply changes to <
                                        {
                                                temp = front[j];             //
swap elements
                                                front[j] = front[j + 1];
                                                front[j + 1] = temp;
                                                flag = 1;               //
indicates that a swap occurred.
                                        }
```

```
                        }
                }


                front.at(0).cd.at(i) += front.at(front.size() -
1).objectives.at(i);
                front.at(front.size() - 1).cd.at(i) =
front.at(front.size() - 1).objectives.at(i);
                if (front.size() == 1)
                {
                        front.at(0).crowding_distance +=
front.at(0).cd.at(i);
                }
                for (int d = 2; d < front.size() - 1; d++)
                {
                        front.at(d).cd.at(i) = front.at(d).cd.at(i) +
(front.at(d + 1).cd.at(i) - front.at(d - 1).cd.at(i)) /
(front.at(front.size() - 1).objectives.at(i) - front.at(0).objectives.at(i));
                        front.at(d).crowding_distance +=
front.at(d).cd.at(i);
                }
        }
    }
    return front;
}

bool NSGA2::crowding_operator(Individual i, Individual j)
{
    if (i.rank < j.rank || ((i.rank == j.rank) && (i.crowding_distance >
j.crowding_distance)))
        return true;
    else
        return false;
}

void NSGA2::create_initial_population()
{
    while (population.population.size() < num_of_individuals)
    {
        population.population.push_back(generate_individual());
        for (int i = 0; i < population.population.size(); i++)
        {
            for (int j = 0; j < population.population.size(); j++)
            {
                if (i < j)
                {
                    double temp =
std::accumulate(grid.begin(), grid.end(), 0.0) -
std::accumulate(population.population.at(j).features.begin(),
population.population.at(j).features.end(), 0.0);
                    while
(population.population.at(i).features == population.population.at(j).features
&& temp < load_to_be_shed)
                    {

    mutate(population.population.at(j));
                        temp =
std::accumulate(grid.begin(), grid.end(), 0.0) -
std::accumulate(population.population.at(j).features.begin(),
population.population.at(j).features.end(), 0.0);
                    }
                }
            }
```

```
			}
		}
}

void NSGA2::calculate_objectives(Individual & individual)
{
		individual.objectives.clear();
		individual.objectives.push_back((load -
std::accumulate(individual.features.begin(), individual.features.end(), 0.0))
- load_to_be_shed);
		std::vector<int> sheded;
		for (int i = 0; i < individual.features.size(); i++)
		{
			if (individual.features.at(i) == 0)
				sheded.push_back(i);
		}
		double pen1 = 0;
		for (int i = 0; i < sheded.size(); i++)
		{
			for (int j = 0; j < previously_sheded.size(); j++)
			{
				if (sheded.at(i) == previously_sheded.at(j).id - 1)
					pen1 += 50 * (j + 1);
				else
					pen1++;
			}
		}
		individual.objectives.push_back(pen1);
		double pen2 = 0;
		individual.medical = 0;
		for (int i = 0; i < sheded.size(); i++)
		{
			for (int j = 0; j < areas.size(); j++)
			{
				if (sheded.at(i) == areas.at(j).id - 1)
				{
					if (areas.at(j).area_type == "Residential" &&
times > 8 && times < 18)
						pen2--;
					if (areas.at(j).area_type == "Economical" &&
times > 8 && times < 18)
						pen2 += 20;
					if (areas.at(j).area_type == "Governmental" &&
times > 8 && times < 15)
						pen2 += 40;
					if (areas.at(j).area_type == "Medical")
					{
						pen2 += 1000000;
						individual.medical++;
					}
				}
			}
		}
		individual.objectives.push_back(pen2);
}

Individual NSGA2::generate_individual()
{
		Individual individual;
		individual.features = grid;
		double load = std::accumulate(individual.features.begin(),
individual.features.end(), 0.0) - std::accumulate(grid.begin(), grid.end(),
0.0);
```

```
	std::vector<int> temp;
	int change = 0;
	int g = 10;
	while (load < load_to_be_shed)
	{
		for (int t = 0; t < temp.size(); t++)
		{
			individual.features.at(temp.at(t)) =
grid.at(temp.at(t));
		}
		temp.clear();
		if (change == 10)
		{
			change = 0;
			g++;
		}
		int genes = std::rand() % (g);
		for (int i = 0; i < genes; i++)
		{
			int r = std::rand() % (grid.size());
			individual.features.at(r) = 0;
			temp.push_back(r);
		}
		load = std::accumulate(grid.begin(), grid.end(), 0.0) -
std::accumulate(individual.features.begin(), individual.features.end(), 0.0);
		change++;
	}
	return individual;
}

Individual NSGA2::tournament()
{
	std::vector<Individual> participants;
	for (int i = 0; i < num_of_tour_participants; i++)
	{
		int r = std::rand() % (population.population.size());
		participants.push_back(population.population.at(r));
	}
	int best = -1;
	for (int i = 0; i < participants.size(); i++)
	{
		if (best == -1 || crowding_operator(participants.at(i),
participants.at(best)))
			best = i;
	}
	return participants.at(best);
}

std::vector<Individual> NSGA2::crossover(Individual a, Individual b)
{
	std::vector<Individual> res;
	Individual child1 = generate_individual();
	Individual child2 = generate_individual();
	int r = std::rand() % (child1.features.size());
	for (int i = 0; i < a.features.size(); i++)
	{
		if (i < r)
		{
			child1.features.at(i) = b.features.at(i);
			child2.features.at(i) = a.features.at(i);
		}
		else {
			child1.features.at(i) = a.features.at(i);
			child2.features.at(i) = b.features.at(i);
```

```
                }
        }
        res.push_back(child1);
        res.push_back(child2);
        return res;
}

void NSGA2::mutate(Individual & child)
{
        for (int i = 0; i < number_of_genes_to_mutate; i++)
        {
                int r = std::rand() % (grid.size());
                child.features.at(r) = child.features.at(r) > 0 ? 0 :
base.at(r);
        }
}

std::vector<Individual> NSGA2::create_children()
{
        std::vector<Individual> children;
        while (children.size() < population.population.size())
        {
                Individual parent1 = tournament();
                Individual parent2 = parent1;
                while (parent1.features != parent2.features)
                {
                        parent2 = tournament();
                }
                std::vector<Individual> temp = crossover(parent1, parent2);
                for (int i = 0; i < temp.size(); i++)
                {
                        mutate(temp.at(i));
                        children.push_back(temp.at(i));
                }
        }

        for (int k = 0; k < children.size(); k++)
        {
                for (int j = 0; j < children.size(); j++)
                {
                        if (k < j)
                        {
                                double temp = std::accumulate(grid.begin(),
grid.end(), 0.0) - std::accumulate(children.at(j).features.begin(),
children.at(j).features.end(), 0.0);
                                while (children.at(k).features ==
children.at(j).features && temp < load_to_be_shed)
                                {
                                        mutate(children.at(j));
                                        temp = std::accumulate(grid.begin(),
grid.end(), 0.0) - std::accumulate(children.at(j).features.begin(),
children.at(j).features.end(), 0.0);
                                }
                        }
                }

        }

        return children;
}
```

8.4 User Interface

8.4.1 Index.html

```
<!DOCTYPE html>
<meta charset="utf-8">
<style type="text/css">

      .node {
    cursor: pointer;
  }

  .overlay{
      background-color:#EEE;
  }

  .node circle {
    fill: #fff;
    stroke: steelblue;
    stroke-width: 1.5px;
  }

  .node text {
    font-size:10px;
    font-family:sans-serif;
  }

  .link {
    fill: none;
    stroke: #ccc;
    stroke-width: 1.5px;
  }

  .additionalParentLink {
    fill: none;
    stroke: blue;
    stroke-width: 1.5px;
  }

  .templink {
    fill: none;
    stroke: red;
    stroke-width: 3px;
  }

  .ghostCircle.show{
      display:block;
  }

  .ghostCircle, .activeDrag .ghostCircle{
       display: none;
  }

</style>
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="dndTree.js"></script>
<body>
    <div id="tree-container"></div>
</body>
</html>
```

8.4.2 Dndtree.js

```
// Get JSON data
//treeJSON =
function load_grid()
{
       d3.select("svg").remove();

       treeJSON = d3.json("4th-Year-Project-master/Implementation/Grid-
Model/grid.json", function(error, treeData) {

    // Calculate total nodes, max label length
    var totalNodes = 0;
    var maxLabelLength = 0;
    // variables for drag/drop
    var selectedNode = null;
    var draggingNode = null;
    // panning variables
    var panSpeed = 200;
    var panBoundary = 20; // Within 20px from edges will pan when dragging.
    // Misc. variables
    var i = 0;
    var duration = 750;
    var root;

    // size of the diagram
    var viewerWidth = $(document).width();
    var viewerHeight = $(document).height();

    var tree = d3.layout.tree()
        .size([viewerHeight, viewerWidth]);

    // define a d3 diagonal projection for use by the node paths later on.
    var diagonal = d3.svg.diagonal()
        .projection(function(d) {
            return [d.y, d.x];
        });

    // A recursive helper function for performing some setup by walking
through all nodes

    function visit(parent, visitFn, childrenFn) {
        if (!parent) return;

        visitFn(parent);

        var children = childrenFn(parent);
        if (children) {
            var count = children.length;
            for (var i = 0; i < count; i++) {
                visit(children[i], visitFn, childrenFn);
            }
        }
    }

    // Call visit function to establish maxLabelLength
    visit(treeData, function(d) {
        totalNodes++;
        maxLabelLength = Math.max(d.name.length, maxLabelLength);

    }, function(d) {
        return d.children && d.children.length > 0 ? d.children : null;
    });
```

```
    // sort the tree according to the node names

    function sortTree() {
        tree.sort(function(a, b) {
            return b.name.toLowerCase() < a.name.toLowerCase() ? 1 : -1;
        });
    }
    // Sort the tree initially incase the JSON isn't in a sorted order.
    sortTree();

    // TODO: Pan function, can be better implemented.

    function pan(domNode, direction) {
        var speed = panSpeed;
        if (panTimer) {
            clearTimeout(panTimer);
            translateCoords = d3.transform(svgGroup.attr("transform"));
            if (direction == 'left' || direction == 'right') {
                translateX = direction == 'left' ?
translateCoords.translate[0] + speed : translateCoords.translate[0] - speed;
                translateY = translateCoords.translate[1];
            } else if (direction == 'up' || direction == 'down') {
                translateX = translateCoords.translate[0];
                translateY = direction == 'up' ? translateCoords.translate[1]
+ speed : translateCoords.translate[1] - speed;
            }
            scaleX = translateCoords.scale[0];
            scaleY = translateCoords.scale[1];
            scale = zoomListener.scale();
            svgGroup.transition().attr("transform", "translate(" + translateX
+ "," + translateY + ")scale(" + scale + ")");
            d3.select(domNode).select('g.node').attr("transform",
"translate(" + translateX + "," + translateY + ")");
            zoomListener.scale(zoomListener.scale());
            zoomListener.translate([translateX, translateY]);
            panTimer = setTimeout(function() {
                pan(domNode, speed, direction);
            }, 50);
        }
    }

    // Define the zoom function for the zoomable tree

    function zoom() {
        svgGroup.attr("transform", "translate(" + d3.event.translate +
")scale(" + d3.event.scale + ")");
    }


    // define the zoomListener which calls the zoom function on the "zoom"
event constrained within the scaleExtents
    var zoomListener = d3.behavior.zoom().scaleExtent([0.1, 3]).on("zoom",
zoom);

    function initiateDrag(d, domNode) {
    }

    // define the baseSvg, attaching a class for styling and the zoomListener
    var baseSvg = d3.select("#tree-container").append("svg")
        .attr("width", viewerWidth)
        .attr("height", viewerHeight)
        .attr("class", "overlay")
```

```
        .call(zoomListener);


    // Define the drag listeners for drag/drop behaviour of nodes.
    dragListener = d3.behavior.drag()
        .on("dragstart", function(d) {
            if (d == root) {
                return;
            }
            dragStarted = true;
            nodes = tree.nodes(d);
            d3.event.sourceEvent.stopPropagation();
            // it's important that we suppress the mouseover event on the
node being dragged. Otherwise it will absorb the mouseover event and the
underlying node will not detect it d3.select(this).attr('pointer-events',
'none');
        })
        .on("drag", function(d) {
            if (d == root) {
                return;
            }
            if (dragStarted) {
                domNode = this;
                initiateDrag(d, domNode);
            }

            // get coords of mouseEvent relative to svg container to allow
for panning
            relCoords = d3.mouse($('svg').get(0));
            if (relCoords[0] < panBoundary) {
                panTimer = true;
                pan(this, 'left');
            } else if (relCoords[0] > ($('svg').width() - panBoundary)) {

                panTimer = true;
                pan(this, 'right');
            } else if (relCoords[1] < panBoundary) {
                panTimer = true;
                pan(this, 'up');
            } else if (relCoords[1] > ($('svg').height() - panBoundary)) {
                panTimer = true;
                pan(this, 'down');
            } else {
                try {
                    clearTimeout(panTimer);
                } catch (e) {

                }
            }

            d.x0 += d3.event.dy;
            d.y0 += d3.event.dx;
            var node = d3.select(this);
            node.attr("transform", "translate(" + d.y0 + "," + d.x0 + ")");
            updateTempConnector();
        }).on("dragend", function(d) {
            if (d == root) {
                return;
            }
            domNode = this;
            if (selectedNode) {
                // now remove the element from the parent, and insert it into
the new elements children
```

```
                var index =
draggingNode.parent.children.indexOf(draggingNode);
                if (index > -1) {
                    draggingNode.parent.children.splice(index, 1);
                }
                if (typeof selectedNode.children !== 'undefined' || typeof
selectedNode._children !== 'undefined') {
                    if (typeof selectedNode.children !== 'undefined') {
                        selectedNode.children.push(draggingNode);
                    } else {
                        selectedNode._children.push(draggingNode);
                    }
                } else {
                    selectedNode.children = [];
                    selectedNode.children.push(draggingNode);
                }
                // Make sure that the node being added to is expanded so user
can see added node is correctly moved
                expand(selectedNode);
                sortTree();
                endDrag();
            } else {
                endDrag();
            }
        });

    function endDrag() {
        selectedNode = null;
        d3.selectAll('.ghostCircle').attr('class', 'ghostCircle');
        d3.select(domNode).attr('class', 'node');
        // now restore the mouseover event or we won't be able to drag a 2nd
time
        d3.select(domNode).select('.ghostCircle').attr('pointer-events', '');
        updateTempConnector();
        if (draggingNode !== null) {
            update(root);
            centerNode(draggingNode);
            draggingNode = null;
        }
    }

    // Helper functions for collapsing and expanding nodes.

    function collapse(d) {
        if (d.children) {
            d._children = d.children;
            d._children.forEach(collapse);
            d.children = null;
        }
    }

    function expand(d) {
        if (d._children) {
            d.children = d._children;
            d.children.forEach(expand);
            d._children = null;
        }
    }

    var overCircle = function(d) {
        selectedNode = d;
        updateTempConnector();
    };
    var outCircle = function(d) {
```

```
        selectedNode = null;
        updateTempConnector();
    };

    // Function to update the temporary connector indicating dragging
affiliation
    var updateTempConnector = function() {
        var data = [];
        if (draggingNode !== null && selectedNode !== null) {
            // have to flip the source coordinates since we did this for the
existing connectors on the original tree
            data = [{
                source: {
                    x: selectedNode.y0,
                    y: selectedNode.x0
                },
                target: {
                    x: draggingNode.y0,
                    y: draggingNode.x0
                }
            }];
        }
        var link = svgGroup.selectAll(".templink").data(data);

        link.enter().append("path")
            .attr("class", "templink")
            .attr("d", d3.svg.diagonal())
            .attr('pointer-events', 'none');

        link.attr("d", d3.svg.diagonal());

        link.exit().remove();
    };

    // Function to center node when clicked/dropped so node doesn't get lost
when collapsing/moving with large amount of children.

    function centerNode(source) {
        scale = zoomListener.scale();
        x = -source.y0;
        y = -source.x0;
        x = x * scale + viewerWidth / 2;
        y = y * scale + viewerHeight / 2;
        d3.select('g').transition()
            .duration(duration)
            .attr("transform", "translate(" + x + "," + y + ")scale(" + scale
+ ")");
        zoomListener.scale(scale);
        zoomListener.translate([x, y]);
    }

    // Toggle children function

    function toggleChildren(d) {
        if (d.children) {
            d._children = d.children;
            d.children = null;
        } else if (d._children) {
            d.children = d._children;
            d._children = null;
        }
        return d;
    }
```

```
    // Toggle children on click.

    function click(d) {
        if (d3.event.defaultPrevented) return; // click suppressed
        d = toggleChildren(d);
        update(d);
        centerNode(d);
    }

    function update(source) {
        // Compute the new height, function counts total children of root
node and sets tree height accordingly.
        // This prevents the layout looking squashed when new nodes are made
visible or looking sparse when nodes are removed
        // This makes the layout more consistent.

             var levelWidth = [1];
        var childCount = function(level, n) {

            if (n.children && n.children.length > 0) {
                if (levelWidth.length <= level + 1) levelWidth.push(0);

                levelWidth[level + 1] += n.children.length;
                n.children.forEach(function(d) {
                    childCount(level + 1, d);
                });
            }
        };
        childCount(0, root);
        var newHeight = d3.max(levelWidth) * 25; // 25 pixels per line
        tree = tree.size([newHeight, viewerWidth]);

        // Compute the new tree layout.
        var nodes = tree.nodes(root).reverse(),
            links = tree.links(nodes);

        // Set widths between levels based on maxLabelLength.
        nodes.forEach(function(d) {
            d.y = (d.depth * (maxLabelLength * 10)); //maxLabelLength * 10px
            // alternatively to keep a fixed scale one can set a fixed depth
per level
            // Normalize for fixed-depth by commenting out below line
            // d.y = (d.depth * 500); //500px per level.
        });

        // Update the nodes…
        node = svgGroup.selectAll("g.node")
            .data(nodes, function(d) {
                return d.id || (d.id = ++i);
            });

        // Enter any new nodes at the parent's previous position.
        var nodeEnter = node.enter().append("g")
            .call(dragListener)
            .attr("class", "node")
            .attr("transform", function(d) {
                return "translate(" + source.y0 + "," + source.x0 + ")";
            })
            .on('click', click);

        nodeEnter.append("circle")
            .attr('class', 'nodeCircle')
            .attr("r", 0)
            .style("fill", function(d) {
```

```
                return d._children ? "lightsteelblue" : "#fff";
            });

        nodeEnter.append("text")
            .attr("x", function(d) {
                return d.children || d._children ? -10 : 10;
            })
            .attr("dy", ".35em")
            .attr('class', 'nodeText')
            .attr("text-anchor", function(d) {
                return d.children || d._children ? "end" : "start";
            })
            .text(function(d) {
                return d.name;
            })
            .style("fill-opacity", 0);

        // phantom node to give us mouseover in a radius around it
        nodeEnter.append("circle")
            .attr('class', 'ghostCircle')
            .attr("r", 30)
            .attr("opacity", 0.2) // change this to zero to hide the target
area
        .style("fill", "red")
            .attr('pointer-events', 'mouseover')
            .on("mouseover", function(node) {
                overCircle(node);
            })
            .on("mouseout", function(node) {
                outCircle(node);
            });

        // Update the text to reflect whether node has children or not.
        node.select('text')
            .attr("x", function(d) {
                return d.children || d._children ? -10 : 10;
            })
            .attr("text-anchor", function(d) {
                return d.children || d._children ? "end" : "start";
            })
            .text(function(d) {
                        if(d.generation)
                                return d.name + "\tload:\t" + d.load +
"MW\tgeneration:\t" + d.generation + "MW";
                        else
                                return d.name + "\tload:\t" + d.load +
"MW";
            });

        // Change the circle fill depending on whether it has children and is
collapsed
        node.select("circle.nodeCircle")
            .attr("r", 4.5)
            .style("fill", function(d) {
                        if(d.load == 0 && d.sheded == 0)
                                return "red";
                        else if(d.load == 0 && d.sheded == 1)
                                return "black";
                        else if(d.sheded == 1)
                                return "orange";
                        else
                                return "yellow";
                return d._children ? "lightsteelblue" : "#fff";
            });
```

```
        // Transition nodes to their new position.
        var nodeUpdate = node.transition()
            .duration(duration)
            .attr("transform", function(d) {
                return "translate(" + d.y + "," + d.x + ")";
            });

        // Fade the text in
        nodeUpdate.select("text")
            .style("fill-opacity", 1);

        // Transition exiting nodes to the parent's new position.
        var nodeExit = node.exit().transition()
            .duration(duration)
            .attr("transform", function(d) {
                return "translate(" + source.y + "," + source.x + ")";
            })
            .remove();

        nodeExit.select("circle")
            .attr("r", 0);

        nodeExit.select("text")
            .style("fill-opacity", 0);

        // Update the links…
        var link = svgGroup.selectAll("path.link")
            .data(links, function(d) {
                return d.target.id;
            });

        // Enter any new links at the parent's previous position.
        link.enter().insert("path", "g")
            .attr("class", "link")
            .attr("d", function(d) {
                var o = {
                    x: source.x0,
                    y: source.y0
                };
                return diagonal({
                    source: o,
                    target: o
                });
            });

        // Transition links to their new position.
        link.transition()
            .duration(duration)
            .attr("d", diagonal);

        // Transition exiting nodes to the parent's new position.
        link.exit().transition()
            .duration(duration)
            .attr("d", function(d) {
                var o = {
                    x: source.x,
                    y: source.y
                };
                return diagonal({
                    source: o,
                    target: o
                });
            })
```

```
            .remove();

        // Stash the old positions for transition.
        nodes.forEach(function(d) {
            d.x0 = d.x;
            d.y0 = d.y;
        });
    }
    // Append a group which holds all nodes and which the zoom Listener can
act upon.
    var svgGroup = baseSvg.append("g");

    // Define the root
    root = treeData;
    root.x0 = viewerHeight / 2;
    root.y0 = 0;

    // Layout the tree initially and center on the root node.
    update(root);
    centerNode(root);

       var couplingParent1 = tree.nodes(root).filter(function(d) {
            return d['name'] === 'cluster';
        })[0];
       var couplingChild1 = tree.nodes(root).filter(function(d) {
            return d['name'] === 'JSONConverter';
        })[0];

       multiParents = [{
                    parent: couplingParent1,
                    child: couplingChild1
                }];

       multiParents.forEach(function(multiPair) {
            svgGroup.append("path", "g")
            .attr("class", "additionalParentLink")
                .attr("d", function() {
                    var oTarget = {
                        x: multiPair.parent.x0,
                        y: multiPair.parent.y0
                    };
                    var oSource = {
                        x: multiPair.child.x0,
                        y: multiPair.child.y0
                    };
                    /*if (multiPair.child.depth ===
multiPair.couplingParent1.depth) {
                        return "M" + oSource.y + " " + oSource.x + " L" +
(oTarget.y + ((Math.abs((oTarget.x - oSource.x))) * 0.25)) + " " + oTarget.x
+ " " + oTarget.y + " " + oTarget.x;
                    }*/
                    return diagonal({
                        source: oSource,
                        target: oTarget
                    });
                });
        });
});
//alert("refreshed");
}
load_grid();
setInterval(load_grid, 20000);
```